UNIT 3 PROGRAMMING TOOLS

Structure

3.0 INTRODUCTION

In this unit of the block, we will look at the programming tools available in UNIX. Historically UNIX has been very strong in this area. There is a rich set of tools and utilities available which make the task of maintaining programs and program sets easier, help in keeping dependencies among programs well documented and allow them to be used for correct generation of the executables with the minimum amount of compilation, and aid in version control. There are also tools which help in debugging and profiling programs. Here we will have a brief look at the C compiler in UNIX and some of the options to it which are commonly used, as well as some related utilities. We will also look at the **make** utility and the source code control system (sccs). These should greatly aid you when you are writing complex programs spread across several files.

Although as in all the other units, we will have to be brutally brief, we will try and at least mention the facilities available so that you can go and look up the relevant utility. It would be much more difficult for you to pore over the massive documentation trying to find whether or not there is a utility available to perform some task.

3.1 OBJECTIVES

In this unit, we will round off the block by looking at an area where again UNIX has been traditionally strong, that of program development tools. Finally we will have a very brief look at the duties of a system administrator and will discuss how to perform some of them. At the end of the unit you should be :

- Familiar with the UNIX C compiler
- Discuss the main C programming tools available in UNIX, like the profiler, verifier and cflow
- Able to use the **make** utility to maintain packages
- Familiar with the Source Code Control System

3.2 THE UNIX C COMPILER

The history of UNIX and C are so closely related that we will here remind you that there are other languages also available under UNIX, but we will look at only the UNIX C compiler here. There are various other C compilers now available under UNIX, but we will not consider any of them. The UNIX C compiler is called cc, and it is available in /bin.

To obtain a running program, you have to compile the source file and create an executable version of it. The source file can be created using any text editor, for example, vi. Suppose you have the following toy program

```
% cat first.c
main(argc,argv)
int argc;
char *argv[];

{
switch (argc)
{
case 1:
printf("No argument; it's a deal\n");
exit(0);
case 2:
printf("One argument\n");
exit(0);
default:
printf("Lots of arguments\n");
}
}
```

This prints a message depending on how many arguments you give it. But how do you make it run? The first thing to note is that the name of the source file must end in ".c" for the compiler to recognise it as a C source file. If the name ends in any other characters the program will not compile at all, even if it is a perfectly correct C program. Now say

```
% cc first.c
```

If you have not made any mistake while keying in the program, the compiler will do its work silently and you will get the system prompt as soon as it is done. When you use the compiler in this fashion, the name of the executable file that it creates is always "a.out". So to run the program say

```
% a.out

No argument; it's a deal
```

and you get your output. Of course it is not always possible to have a single name for the executable file. You might want to have several executables at the same time or you might want to compile programs one after another. You could either rename your executable

```
% mv a.out first
```

or have the compiler itself produce the executable with a different name

```
% cc first.c -o first
```

The -o flag tells the C compiler to give the executable the name which follows the flag. So you now have an executable file called first which you can run. In both the cases above, the compiler makes the file with execute permission and so you do not have to change the permission bits, as was necessary when entering shell scripts.

What happens when you have programs spread across more than file? You then have to create object modules from both files and link them together. Suppose you have your source programs in two files part1.c and part2.c, and you wish to call your final executable file opus. You can do this as follows

```
% cc -c part1.c

% cc -c part2.c
```

```
% cc part1.o part2.o -o opus
```

The first two lines create the object code in files with a ".o" suffix instead of ".c". So files called part1.o and part2.o will get created. The third line creates the executable. As you know this process involves linking the object modules with the various library files. The main C library is called libc.a and it is located in the /lib directory. When you use cc it is included by default. But if you want to use the linker explicitly, you can do so with the ld command. We do not consider this here because for most purposes the cc command will suffice. If you use ld, however, you must specify the startup files and all libraries. These tiresome details get taken care of by the cc command, so it is not necessary to go into those matters. You can also create the executable opus with one cc command

```
% cc part1.c part2.c -o opus
```

part1.c:

part2.c:

This will create the object files part1.o and part2.o and link them together as well as with the standard C library to produce the executable file opus.

Sometimes you want to link some other library file to the programs you write. For example you might have used some functions like sqrt which are available only in the math library "libm.a". This file is in the /lib directory. In general, the library files are available in this directory and they have the suffix ".a" because they are archive files created with the ar command. To have this library included while linking, you have to give the following command

```
% cc part1.c part2.c -o opus -lm
```

In general to include a library file in /lib/libname.a, you have to say -lname on the command line. If there is more than one library file to be put you simply give that as well. Libraries other than libc.a, libx.a and libm.a might be required for different software packages. Thus if you want to compile a curses program (that is, one using curses features) it will not be enough to use these libraries and you will need to include the curses library as well.

Similarly you could be writing a software package of your own, wherein you might be using several specialised functions. Instead of making the source available, you could compile the functions and keep them in a library of your own. Then anyone wanting to use your library functions would need to link in your library. You will need to refer to the documentation of UNIX to understand the process properly and avoid pitfalls, but we mention this here to tell you that there is nothing sacrosanct about the standard libraries and you can create your own libraries as well. To create an archive file which can be used as a library you will need to take the help of the ar command. We will not discuss it here though, except the following

```
% ar t /lib/libc.a
```

will give you the list of all functions that the library contains. The t option stands for table of contents. There are options to add and extract elements to and from the archive, to create the archive the first time, and so on.

Now so far we have been talking of compiling programs to create an executable file. You can also get assembly language output from the C-compiler

```
% cc -S first.c
```

will produce a file called "first.s" with the assembly language output in it. You can link assembly language files with C files on the command line using the cc command just as you can link object code modules.

If you have preprocessor statements in your C source program and you want to view their expansion, you can say

```
% cc -P first.c
```

which will only preprocess the statements in the file and produce an output file first.i. Another useful flag is the -D flag for debug. Using this you can compile the program including the statements which are to be executed if the variable specified after D is defined.

All the above assumed a perfect program which compiles without any difficulty. What happens when you have errors? The compiler often ends up giving you a flurry of errors even if you have made only a small mistake in your program. Thus a missing semicolon at the end of a statement will make the rest of the program quite unintelligible to the compiler and you will get a large number of errors. When the compiler feels that there are too many errors, it refuses to compile any further and stops. Otherwise it tries to carry on as long as it can.

The error messages after the first one are not too reliable because as we just saw any error affects the compilation of the rest of the code. So while trying to get a program to compile it is wise to start from the first error and try to remove it. You will often find that this reduces the number of errors drastically. However it might also result in other errors coming into view. The error message for the first error is usually helpful enough in diagnosing the problem.

The errors obtained in the linking phase usually have to do with unsatisfied external references. When you get such an error you should make sure that you have included the appropriate header files and have linked in the required libraries.

3.3 SOME OTHER TOOLS

Apart from the C compiler proper, there are many software development tools which can help you in various ways. We will look at some of them briefly. In UNIX there are so many of these that a novice can easily get confused on just what aids are available. The short discussion on the tools here is intended to serve as a guide to availability rather than a detailed description of the tools themselves. You should make use of whatever is appropriate to your needs for you can easily increase your productivity, reduce drudgery and decrease the errors and inefficiencies in your programs by using these facilities judiciously.

3.3.1 Lint - The C Verifier

While the C compiler produces executable code, it is not very strict on things like type checking, unreachable code, return values and so on. And of course it does not worry about portability. However all of these are issues which should be looked at in a program.

For this there is a utility called lint. It is usually described as a tool to pick out the pieces of fluff from a C program. Although lint does not compile the program or produce any code, it checks the program much more strictly than the compiler does. Some of the things it looks for are unused variables and functions, unused return values from functions, questionable constructs (which can often become bugs), constructs which can make the code unportable and other such matters.

Now the thing is that lint is very strict. For example, if you use a printf in your program and do not use its return value, lint will complain:

function returns a value which is always ignored

You do not really need to use the return value of printf usually, and you can therefore ignore this warning from lint. In fact, the messages from lint are so numerous that there are options to lint to suppress certain kinds of checks, and also to perform others that are not done by default. However, lint is particularly useful in the portions of programs involving pointers, and can save you a lot of grief there.

3.3.2 Program Profiles

Let us now take a brief look at how you can analyse the efficiency of a program and maybe improve its performance. To see how long a program takes to execute, you can use the utility /bin/time

```
% time first /dev/null
    real    38.3
    user    37.4
    sys     0.1
```

We redirect all output to the null device, which amounts to discarding it. The reason for this is that we do not want to confuse matters by looking at the time taken for print statements to be executed. As this is a trivial program, it is not a very interesting candidate for a time command. You can write something more appropriate to test out, or run the command on a system utility like grep or sed. Note also that time is a command that takes a command as an argument.

The times given are easy enough to interpret. The real figure is the actual calendar time that elapses while the command is being executed. This will depend on the load on the system if it is a multi-user installation. Even if you are the only user, there is still the possibility that you have other processes running in the background. Many system daemons are also started up at boot time and they can also slow down the execution of your program. So to get a consistent comparison for different programs, you need to run the system in a standard state. For example test out the times in single user mode with no background processes running. Even here the time command is not very accurate and its granularity can affect the results marginally. You can run a command repeatedly, say 100 times, and measure the time taken for that. This will give you a more accurate idea of the time taken for a single run.

The user figure given by the command is the time spent by the system in executing your program code. This should not change much (except for the granularity of the time command itself) with variations in system load. Likewise, the sys figure tells you the amount of time spent in executing UNIX system code for your program. This time too does not depend on external load.

What would you do if you had written a complex program which did not seem to work as fast as you had expected? You could try improving your code to make it more efficient. While this might not always be possible, it usually is, because while you wrote the program the first time around, you probably spent your time more on making it correct and working than on efficiency issues. As it is, a few percentage points do not usually matter as far as most programs are concerned, and major improvements in efficiency come from better algorithms rather than anything else.

But sometimes it is important to speed up your program by that 10% ratio, or your program is running so slow that you suspect some gross inefficiency in the coding itself. You can then analyse the profile of the program and identify the problem spots for attention. For this you need to compile the program with the -p (for profile) option

```
% cc -p first.c -o first
```

The code now generated is longer because some extra code to draw on the UNIX system monitor routine is also included. The profile data is stored in a file called "mon.out" when the program is run and a utility called prof is then run to interpret the results.

```
% prof first
```

This produces an output giving for each function in your program, the percentage of the total time spent executing that function, the cumulative time spent executing it, the number of calls to the function and the time spent per call. From this data you can quickly see where an improvement could be most useful. If only 2% of the time is spent on a function, then even a drastic improvement there will not make much difference overall (0.4% total if it gets improved by 20%). But if 38% of the time is spent on some other function, then a 5% improvement here will reduce the overall time by 1.9%, and so this is the function to attack.

There are a few things to keep in mind when you use prof to check out a program's profile in an endeavour to make it more efficient. The profile and the analysis are valid only for the data set the program was run on, and if that was a very uncommon data set then it might not be much use trying to improve the performance of your program there. There might be tradeoffs required to be done for different data sets as an improvement in one place might result in difficulties in some other case. You will then have to decide which is more important.

Also when you run the prof utility, it works on the latest "mon.out" available. This must be the one obtained by running the program to be profiled. If you run it on some other "mon.out" the results will be meaningless. While it is easy to see when the prof gives you strange functions (so that you know straightaway if you have used the file from some other program), it is more difficult to be sure that you are using the proper data set. So it is best to run your program before you call up prof, so that you know which data set you have used.

3.3.3 Program Listings

There is a utility called cb (for C beautifier) that beautifies listings. Different programmers have different ways of entering programs so that the listings look different. For example look at the following program fragment

```
while (a  c)
    {
        if (x == y % 3)
        {
            a += b;
            c += d;
        }
        ...
    }
```

This is one style of indenting loops and conditions. But some people are not so meticulous and the same code might have been entered like this

```
while  (a  c)
    {
        if (x == y % 3)
        {
            a += b;
            c += d;
        }
        ...
    }
```

This has no indentation and uses the K & R style of braces after loops. In C white space has no significance except in literals and so the two fragments are equivalent as far as the compiler is concerned. But for a person used to the indented style the second example can be difficult to decipher. Other programmers might have even different styles. For example some people put more than one statement on a line

```
while (a  c) {
if (x == y % 3) { a += b; c += d;}...}
```

This is the most difficult to read of the lot. If you are faced with maintaining or modifying such code, you can use cb to produce a beautiful listing from either of the last two examples. Then you can at least get down to your task with a readable layout before you. It can certainly make your task easier.

Beauty is a term difficult to define, and the layout cb produces might not be in keeping with your tastes. In that case there is no way out except to write your own version, but even if you use the standard cb you will at least get a consistent style from any input. You only have to say

```
% cb hard.c  easy.c
```

cb writes to the standard output and this necessitates redirecting the result if a disk file output is needed. Also cb does not work well unless the program compiles, because syntax errors can confuse cb. For example if you have said it instead of if, cb can do nothing about it. It will not indent the code following the it as it would have had it encountered an if.

cb has an option to produce K & R style of code with the opening brace following a loop coming on the same line after one space. The default is to put the brace on the next line.

This utility is useful in enforcing layout standards. If you are working on a large project with different programmers producing different parts of the package, you can pass each program through cb before accepting it as part of the package. This way all listings will look alike in the consolidated package.

3.3.4 Cross References and Program Flow

When a program is split across several files, it often becomes difficult to ascertain where a particular variable or function occurs. The same thing can be true if you have a large program in one file. If you want to find out only one variable it is easy and you already know several ways of locating the variable name in the files, say by using grep or vi. But that does not help you produce a list of all variables or functions.

There is a utility called cxref to do just this. It produces a cross reference table for all C source files given to it as arguments. The output is written to the standard output and has to be redirected to a disk file if needed.

The output consists of four columns. The first column contains the variable names. Function names are also mentioned and are followed by a pair of parentheses to show that they are functions. The second column gives the names of the files in which a reference to the variable or the function occurs. The third column gives the names of the functions which contain the reference. If more than one function in the file uses the variable or function, then there is one output line for each such reference. The last line contains a list of line numbers in the source file where the reference occurs. The line number of the line containing the declaration of the variable is preceded by an asterisk. You should look up the UNIX documentation for more details on cxref and the options available to control the output it produces.

A somewhat similar problem is that of getting a picture of the flow of control in a set of programs constituting a package. The cross reference utility cxref tells you which functions call a given function. To find out which functions a given function calls, you can use the cflow utility. This gives you a flow graph of the program set, telling you all the functions called by every function starting from main(). You can also obtain the reverse picture by using the proper option to cflow. The reverse flow graph will tell you which functions call the functions in the program set. Thus this utility is of value in understanding at a glance the flow control of the set of programs in question.

3.3.5 Other Tools

We will be considering only two more sets of tools in some detail. These are make for efficiently compiling sets of programs to create the final executable and the source code control system (SCCS) which is invaluable in maintaining the source code for different versions of programs economically and easily. Before that we will very briefly mention some tools and utilities and what they do so that you can obtain more details from the UNIX documentation whenever you need to.

The utilities we have looked at here like cxref, cflow, cb and cprof all assume that there is a working program set which you want to analyse. But when it comes to actually debugging a program, these utilities are not of any help. In UNIX we have debuggers called adb (absolute debugger) and sdb (symbolic debugger) which help you to examine programs. They have the facilities usually associated with a debugger like looking at the values of variables, setting breakpoints, stepping through a program and so on. One of the two is quite likely to be available at your installation.

If you are working in C, you can also use the ctrace utility to get a trace of what your program is doing. It gives a very large quantity of output. However, it is possible to arrange to trace only certain parts of the program.

There are a couple of tools that are useful if you want to write a compiler or text processing system. These are lex and yacc. Lex can be very helpful in writing the lexical analyser for a compiler, since you can easily tell it which patterns to recognise as tokens. These tokens can be then passed onto the syntax analysis phase of the compiler.

Yacc is a tool for generating a parser for a compiler. It can be made to work in cooperation with lex to accept the tokens it finds and use them for syntax analysis. You can also specify actions to be taken when a particular pattern is found. The language can be described in terms of rules in BNF. Yacc executes an LALR(1) parsing algorithm. Apart from full fledged languages, you can use yacc and lex to write other programs, for example a calculator.

3.4 MAINTAINING PROGRAMS

If you have a large number of programs to manage, the task can become quite tedious, especially when you are in a development environment and recompilations are frequent. You would want to see that you compile as little as possible, for in this process compilation time is of significance to you. At the same time you want to be sure that you do not end up with executable programs made up of inconsistent versions of program files.

Suppose you have two executable programs called prog1 and prog2. Also assume that prog1 is made up of components spread across two files part1 and part2, while prog2 is in a file by itself. Now whenever part1 or part2 are changed, the program prog1 becomes outdated and needs to be recompiled. However prog2 is not affected. Likewise a change in prog2 does not affect prog1, which remains current. So you can see that different executables are affected by changes in different program files. It would be therefore wasteful to compile every file every time a change was made to one of them. For example, in the above case, if we compiled prog2 again after a change to part1, it would be an unnecessary activity because prog2 is not connected to part1 at all. In this simple case this might seem easy to do but that is not so when you are dealing with dozens of programs spread across scores of files.

UNIX has a utility called **make** which makes this task much easier. It is all but indispensable when maintaining large programs. It can also be of assistance in maintaining text other than program source files.

Make works by using certain rules to decide how to create the executable program from the source files. If you think about this you will see that the process involves three entities. The first is what to produce as the final program. These are called targets by **make.** In our simple example earlier, prog1 and prog2 are targets because they are the executables to be produced. The second entity is the rules to use to decide how to create the targets from the source files. The third part is which source files to use to create which executables — this information is called a dependency and has to be specified by the user. Here also comes the question of when to use which source file. In general **make** relies heavily on the time stamp of the files to decide whether or not the target needs to be recreated. If a source file has been modified later than the target, it means that the target is out of date and needs to be recreated. Other components of the target need not be remade.

Let us see a small example of how dependencies are specified. We will take our earlier case involving prog1 (consisting of part1 and part2) and prog2 (with only one source file). These dependencies are put into a file using any text editor like vi. We will later explain how **make** uses this file.

```
% cat makefile
# A simple makefile for making prog1 and prog2
prog1:  part1.o part2.o
        cc -o prog1 part1.o part2.o
part1.o: part1.c
        cc -c part1.c
part2.o: part2.c
        cc -c part2.c
prog2:  prog2.c
        cc -o prog2 prog2.c
```

You can annotate a makefile by using a # at the beginning of a line to indicate a comment. You then indicate each target followed by a colon. Then put a tab character (not a space) and mention all the prerequisites for creating the target. Thus the line specifies a dependency of the target on its prerequisites. The next line gives the rule or method used to create the target from its components.

Thus the first target here is prog1, which depends on part1.o and part2.o. These are to be created by compiling the respective .c files. The second target is prog2 which depends only on prog2.c and is made by compiling it.

To run the **make** command just say:

```
% make
cc -c part1.c
cc -c part2.c
cc -o prog1 part1.o part2.o
```

As **make** executes, it shows you what it is doing. By default, **make** looks for its dependencies in a file called makefile, and makes the first target mentioned there. If you mention some other target given in the makefile, that target is the one that gets made. If makefile does not exist, it looks for Makefile. You can also put the commands into any file you want and ask **make** to look at that file by using the -f flag like this

```
% make -f mkfile
```

will tell **make** to look at mkfile for its commands. A file which contains commands for **make** to follow is commonly called a makefile. Thus here mkfile is a makefile. You can also tell make to produce only some of the targets like this

```
% make -f mkfile prog2 part2.o
cc -o prog2 prog2.c
cc -c part2.c
```

Let us say we now make some change to part2.c and want to get our programs upto date. Simply say

```
% make -f mkfile
cc -c part2.c
cc -o prog1 part1.o part2.o
```

and **make** will recompile part2.o and prog1. It will not compile prog2 or part1.o. As mentioned before, this is done solely based on the date of modification of the files concerned. In fact if you invoke make again it will not compile anything at all.

```
% make -f mkfile prog1
'prog1' is up to date
```

Although make will do a faithful job of keeping our programs upto date, it is not uncommon to need reassurance that everything is indeed fine. To be sure of that, you might sometimes want to recompile everything and remove all doubt about the correctness of the executable. You can do that by changing the modification date of each source file to the current system date. Just say

```
% touch *.c
```

and all C files will become more recent than the object files or the executable. If you now call up make, it will think that the executable is outdated and will compile everything. Actually the touch command can also be used to give the files any date you want to, but giving it the system time in this fashion is the easiest way of achieving what you want.

We will now look at a slightly different makefile contained in a file also called makefile, so that we do not have to use the -f flag at all. Here we have a program called prog which depends on three parts, part1.c, part2.c and part3.c. Of these part1.c has an include file called part1.h. How do we ensure a minimal recompilation to create the executable prog?

```
% cat makefile
prog:    part1.o part2.o part3.o
         cc part1.o part2.o part3.o -o prog
part1.o: part1.c part1.h
         cc -c part1.c
part2.o: part2.c
         cc -c part2.c
```

```
part3.o: part3.c
        cc -c part3.c
```

This is an elaborate makefile, and is actually longer than needed because make has some built in knowledge which you can use. Thus make knows that a .o depends on the .c of the same name and that the creation has to be done with a cc command. Actually make knows about other tools like lex, yacc and sccs as well. These matters should be understood by looking at the documentation for make. Using this knowledge, you can shorten the makefile to the following

```
% cat makefile
prog:   part1.o part2.o part3.o
        cc part1.o part2.o part3.o -o prog
part1.o: part1.c part1.h
        cc -c part1.c
```

If you now execute this makefile, you will find that the creation of the .o files for part2.o and part3.o is done using the -O (for optimise) flag of the C compiler. This is because the built in rules for make are such that the compiler is called up with the optimiser whenever you want to go from a .c to a .o or to an executable. It is possible to look at as well as change these built in rules to suit your taste. For example, if you want the compilation to proceed without calling up the optimiser, it is possible to specify the method of proceeding from a .c to a .o file and there mention the cc command without the -O flag.

A useful option to make is the -n option which shows what make would have otherwise done, but does not actually execute the commands. This can also give you an indication of the amount of work involved in constructing a target. It is also useful for debugging a makefile. Do not be surprised at the idea, for although we have taken absurdly simple examples here, in practice a makefile can be quite complex. A mistake in constructing the makefile can be disastrous at worst and misleading at best. So you should take great care while specifying your makefile, but once that is done, you can relax as far as compiling your package is concerned.

We shall again mention that make can be used not only for software packages, but whenever you have a situation of targets depending on some constituents and automatically producible from them according to some well defined rules. You can thus use make for various other purposes like documentation.

There are many features of make that we have not discussed here. Thus you can use macro definitions in a makefile to avoid having to repeat a large amount of text again and again. You can specify certain intermediate targets as precious so that a previous target is not removed unless the next one has got created. You can invoke shell commands from a makefile and so it can be made to give messages after performing certain tasks. You would do well to study the features and capabilities of this command in detail if you have anything to do with producing software packages.

3.5 THE SOURCE CODE CONTROL SYSTEM

The Source Code Control System, usually called SCCS, is a set of utilities that allow you to handle version control of software packages and other text files, such as those used in documentation. Usually software packages are developed by a group of people. It is necessary to see here that the same program is not being edited by more than one person at the same time. Also it is necessary to put together the latest (or some other known) versions of the constituent source files to create the executable program. This task might seem simple but quickly becomes non-trivial as the number of programs and programmers increases.

Sometimes it happens that a change to a program turns out to be a dead end and it is necessary to backtrack to a previous version and try some other approach. This might not be the immediately previous version. Clearly it is desirable to have access to all the versions of each program that is being written. This could be done by keeping each version separately, but that would require a lot of disk space apart from being very cumbersome, for how would you quickly lay your hands on a given version number? It is here that SCCS provides a way out by providing access to all the versions of every file that you want to maintain. Also, it provides a locking mechanism whereby you cannot mistakenly give out a program to more

than person at the same time for editing. Thus it helps in maintaining consistent versions of files. The make command described earlier understands SCCS commands and knows how to extract a file from an SCCS file.

The basic mechanism that SCCS uses is to keep the initial version of the file and also the set of changes to it until the next version is finalised. At any point you can assert that the current state of the file is to be defined as a version. This version is then given a number for identification. You can then continue to make changes to the file again until you feel that another version can be announced. At any time you can get back any of the versions including the initial version with which you began the file. SCCS is thus useful even if you are the only one working on the package.

A change in SCCS terminology is called a delta. To use its capabilities to advantage, you should include version control statements in your source files. These are certain keywords used for identification by SCCS and are indicated in the source file by certain upper case letters enclosed in % signs. While SCCS will work without these keywords, it will complain (only a warning) all the time about their absence and you will lose the benefits they provide. We will see later the advantages of having the keywords present. There are a great many keywords but probably only a few are of real use. For example, a useful keyword is %I%, which gets expanded in the appropriate circumstances to the version number of the version in question. Similarly, %E% shows up as the date the version was created.

With this background, let us take a quick look at the SCCS commands which allow us to do all that we have talked of. As usual, while we will not be able to get into much detail, we will certainly talk of the main features and options. You are referred to the UNIX documentation for a comprehensive discussion of SCCS and its commands.

3.5.1 Initialising a File

It would be a good idea to have all your source files in a separate directory, say ~/src. While this makes sense in any context, it is even more useful when you start using SCCS, because the system makes several auxiliary files for its own use and you might otherwise get distracted by the large number of files that you find lying around. So

```
% mkdir ~/src
```

```
% chmod 775 ~/src
```

This we do to give other members of our group write permission on the directory so that they can create files in it. Without this ability only you would be able to create and hand out files. That would be a good idea if there is to be a single person designated to perform this task of version control. Then every programmer who needed a program could mail a request to you and receive the program by mail, if eligible and if the program was available. Of course, other distribution mechanisms could be used as well.

Now let us say that we are developing the program referred to earlier in the discussion on make, prog. This consists of three programs part1.c, part2.c and part3.c, and one header file part1.h. To create the initial SCCS versions of your files, you need to use the admin command

```
% admin s.part1.c -ipart1.c
% admin s.part1.h -ipart1.h
% admin s.part2.c -ipart2.c
% admin s.part3.c -ipart3.c
```

The command can work on only one file at a time and to create all our initial files, we have to invoke the command again and again. You could also use a shell script to do the job if you have a large number of files.

The SCCS files have to start with s., but this prefix does not get put automatically. So you will get an error if you neglect or forget to put this in. The -i flag to the admin command tells it to initialise the SCCS file, usually called the s file of the source file, with the contents of the file specified after it. As soon as you have done this, you can and should delete the original files. This is because the s files are now the definitive copies and having the older files around is only going to cause confusion at some point of time. If you want to be safe, you should backup your s files. The contents of your src directory will now be as under

```
% ls -x
s.part1.c       s.part1.h       s.part2.c       s.part3.c
```

The s files are text files and you can cat or edit them. However, you are strongly advised not to tamper with an s file directly (which means you should not edit it), because the s file contains information put in by SCCS. If you mess around with that, your file might become unusable and that could be catastrophic, resulting in losing the entire contents. Always use SCCS tools to access or alter the contents of your s files.

The admin command has several other options. You can look at the documentation for an explanation of what they are and what they do. To get brief help on SCCS commands or error codes, you can use the help command. This command gives the syntax of any SCCS commands given to it as arguments. It also gives a brief explanation of the error codes that SCCS commands give whenever there is some error. The codes are not all errors and some can be mere warnings. For example cm7 is the code you get when your file does not contain any identification keywords.

If you want to be sure that your file has the keywords, you can use the -f flag when you are initialising the s file. This will make SCCS treat the absence of keywords as an error rather than a mere warning, and will thus force you to put the keywords in. But you might forget to use the -f flag when you use admin on the file. To get around that you will need to have a front end to SCCS or at least to the admin command. You can easily create such a front end using a shell script, which makes admin run with the -f flag.

You will have noticed by now that the s files are created with read permission for all. How then does one alter a source file? To do that you need to use the get and delta commands described in the next section. The s files have only read permission because they are not intended to be worked upon directly by the user, as explained earlier.

3.5.2 Examining and Altering Files

You know that the s files have the whole history of the file from the time it was initialised to the present. This includes all the changes made to any of its versions. The version numbers start from 1.1 and go on from there. Thus you have version 1.2, 1.3 and so on. You can also go back to say version 1.2 and create another branch there, which will be numbered as 1.2.1, 1.2.2 and so on. One more level of this is permissible in SCCS so that you can have a version number consisting of four numbers. Each number can go upto 9999. It will thus be clear that there can be a large number of different versions stored in the same s file. How does one examine a particular version?

To get any version you have to use the get command:

```
% get s.part1.c

Retrieved:

1.7

39 lines
```

This creates a read only file called part1.c in the current directory, which will be the latest version available of the file in question. Thus in this case, the latest version is version 1.7. If you want to get an earlier version like 1.4, you can use the -r flag to the command

```
% get -r1.4 s.part1.c

Retrieved:

1.4

42 lines
```

An earlier version of a file could be larger and it is unlikely you are surprised at this happening. A change to a file can be an addition or a deletion to the file, so a succeeding version can be longer or shorter than the previous one.

You can supply several s files as an argument to the get command and from each of them the corresponding source file will be extracted. Thus you could say

```
% get s.part1.c s.part2.c s.part3.c s.part1.h
```

```
Retrieved:
s.part1.c
1.7
39 lines

s.part2.c
1.3
56 lines

s.part3.c
1.6
98 lines

s.part1.h
1.3
45 lines
```

Thus you can see that the get command is more garrulous than most UNIX commands are. It tells you the version number and the number of lines in each file that it extracts. All of these will be in your current directory but as read only files.

What do you do if you want to alter some of these files? You should never change the mode of the files and start editing them. This is likely to confuse SCCS and can result in difficulty. The proper way to edit a version of a file is to obtain it explicitly for editing with the -e option

```
% get -e s.part1.c
Retrieved:
1.7
new delta 1.8
39 lines
```

This means that you now have version 1.7 of the file part1.c for editing, and that the next change you make (the delta) will result in version 1.8. There is a shorthand available if you want to extract several files using get

```
% cd
% get src
```

This will extract the latest version from all the s files in the directory and place them in the current directory. But you might find it more manageable to extract the files as you need them rather than all at once.

What happens if you do a get with different version numbers on the same s file one after the other? Thus

```
% get s.part1.c
Retrieved:
1.7
39 lines
% get -r1.4 s.part1.c
Retrieved:
1.4
42 lines
```

There is no difficulty. Even though each file that get extracted (this is called the g file) is read

only, the first g file gets silently overwritten by the next one. This is allowed by SCCS because there is nothing lost. Even if you have made a mistake, you only have to get the g file you want. Since no write can be done to the files, there is no danger of data getting lost.

On the other hand, if there is a writeable g file in the directory, SCCS will not allow any get on the corresponding s file. This prevents anyone from reading any version of a file if there is an edit being done on any other version. It also prevents one from editing any version of a file if somebody else is doing so.

You cannot circumvent this protection by doing the get from a different directory. Whenever a get -e is done on an s file, SCCS creates a file called a p file. If a p file exists, no get on the s file can be done. The p file is removed only if the changes are installed in the s file using the delta command to be discussed shortly. Needless to say you should not tamper with a p file yourself and should let SCCS manage the mechanism.

If you only want to look at a given version of a file without altering it or even making a g file, you can say

% get -p -s -r1.4 s.part1.c

The -s flag suppresses the message from get about the version number and the number of lines in it. The -p flag sends the version extracted from the s file to the standard output without creating any g file. This can be redirected to a file or piped to the lp command for printing on the printer. In this way you can obtain any version without disturbing the SCCS mechanism, because this operation will not be construed as a change to the s file.

Once you have the g file available to you for editing, you can use any text editor to change the file, as usual. These changes typically consist of bug fixes, or additions to the file to provide more features, but can certainly be anything else that is needed. You can then test the changes you have made and when you are satisfied, it is time to install these changes into a new version in the s file. This is done by the delta command

% delta s.part1.c

comments? Removed unused variables. Provided for a test to\ reject temperatures below absolute zero.

1.8

14 inserted

3 deleted

36 unchanged

As you can see, the command asks you to enter comments. You should enter something which explains the nature of changes you have made in this version. If you have more than one line of comments to enter you have to escape all but the last newline with a backslash character. You can omit any comments if you wish by just typing a **in response to the prompt from the delta command.**

The command then tells you the version number of the version you have just installed. It is possible to tell delta the number by which the version should be known, but we do not consider such subtleties here. You are also told the nature of changes you made to the previous version to arrive at this version. Thus it tells you how many lines were deleted, how many were added and how many were left unchanged.

When the new version is safely installed in the s file, the delta command removes the g file. Do not be alarmed at this, because your g file can be easily retrieved from the s file by using the get command, as you have already seen. If the g file were to be kept lying around, you might have mistakenly edited it further and this would lead to confusion. Removing the file is thus useful from the point of view of integrity.

The delta command also removes the p file so that it is now permissible for somebody to invoke any version of the s file for editing.

You can see that with various get and delta operations (with the assumption that some changes are made at each such get) the original form of the file can become quite different

after some time. This is not unusual in a programming or documentation environment. If you have the s file and want a history of the changes made to it over the weeks, you can say

```
% prs s.part1.c
```

This gives on the standard output a report giving each version number in the file, the login id of the person who made the change and the comments put in while issuing the delta command. This can be useful in tracing the evolution of a version.

3.5.3 Identification Keywords

We saw earlier the use of identification keywords. SCCS normally expects you to use these keywords as a documentation aid, because these automatically get converted to the actual values whenever you extract a read only copy of the file. For example we have seen that %I% gets replaced by the version number and %E% by the date of the latest delta. Similarly %Z% is a keyword that gets replaced by a special identification string used by the what command and %M% gets replaced by the name of the file. While SCCS will work even if the keywords are absent, it treats their absence as a warning. So some commands like get and delta will complain about this.

'You might be wondering what purpose is served by having these keywords. One of the useful SCCS commands is what, which searches files for every occurrence of a special identification pattern. This pattern is the one which the keyword %Z% expands to and is usually the string @(#). This pattern put in source code will also appear in object code and in the executable. So if you run the what command on an executable you will get to know all its component pieces with whatever other information was put there while writing the code. You can try the what command on some system utilities and commands, for instance

```
% what /bin/ls
```

By using what, you can be sure of the version number of all the components of your executable, thus guarding against mistakes caused by a mix-up in versions.

Another useful identification keyword is %C%, which expands to the line number on which it appears. This can be helpful in debugging.

3.5.4 Miscellaneous Commands

The make command knows about SCCS and you can use this to make sure that when an executable depends on different source files, you have the latest versions of those files. Thus if prog.c depends on part1.c and part2.c, you can have a makefile like:

```
prog:   part1.o part2.o
        cc part1.o part2.o -o prog
part1.o: part1.c part1.h
        cc -c part1.c
part2.o: part2.c
        cc -c part2.c
part1.c: s.part1.c
        get s.part1.c
part2.c: s.part2.c
        get s.part2.c
```

However, you do not have to do this, because whenever make cannot find a file like part1.c, it automatically looks for a file s.part1.c and if it finds that, it knows that a get has to be done. So the last two dependencies, of part1.c and part2.c can be omitted. In fact, if make does not find a makefile or a Makefile, it will look for the corresponding s files and get the makefile from there. The get done by make is with the -p option so that no file is created permanently and after compilation there are no extra files lying around. You would have noticed that the get ensures that the latest version is the one picked up for compilation. This is just another small example of how versatile the make command is.

Apart from the few commands of SCCS we have covered here, there are many others. We will only mention the capabilities, because a discussion of all the commands with all their options will be lengthy. One idea we have mentioned already is to create shell scripts to

serve as front ends to the SCCS commands, so that it becomes easier to use them without making mistakes.

With SCCS you can change the comment entry in an s file, remove a delta from it, combine deltas and find out the differences between any two versions of an s file. You can also restrict access to a particular set of users and lock versions so that they cannot be changed.

Finally you should remember that SCCS can be used for non- programming projects like documentation tasks. As mentioned earlier, even make can be a useful tool here. With this brief introduction to program development tools, we move on to our final topic, System Administration.

Check Your Progress

1. Suppose someone is running a C program located in your directory. Try compiling the program at that time with the same output name. What happens?

   .....

   .....

   .....

2. Look up the documentation and see how to change the built in rules of make.

   .....

   .....

3. Measure the difference in compilation time and execution time for a large program, with and without optimization.

   .....

   .....

4. You have a large makefile that you use to maintain a package. One fine morning you find after working on your package that during the night the system date had been set by somebody to be in

   (a) the past

   (b) the future

   You are not sure which files are affected. What is the impact on your makefile? How do you recover, after setting the system date right?

   .....

   .....

   .....

3.6 SUMMARY

In this unit we looked at the C compiler, because C is so very closely related with the development of UNIX itself. We repeat that this does not mean that there are no other languages supported under UNIX. We looked at other tools like the verifier, the profiler, the flow analyser, the cross referencer and some more.

The make utility makes it easy to maintain large software packages and SCCS allows us to keep track of various versions easily. Together they amount to a powerful set of tools. As always, the real power of the tools lies not in isolation but in togetherness. The philosophy of UNIX is that small things used in cooperation with one another can be very powerful.

3.7 MODEL ANSWERS

Check Your Progress

1. Since the program is running, UNIX will complain saying "Text file busy" and will stop the compilation. You will need to compile under some other name or wait until the program completes.

2. No model answer.

3. Optimisation will mean slower compilation but quicker execution, in general.

4. (a) No model answer.(b) If the system date is put in the future, it will not affect your work. When the date is put right, do a

   % touch filename

   for all filenames which were affected (this you can see with a directory listing). Then you can continue to work.