

Lesson Objectives

- CC
- MAKE
- SVN
- Introduction to System administration



9.1: CC Command

Contents

- **UNIX system compiler for C.**
- **CC compiles the file. If no errors are found, an executable version of the file is created. The default name of the executable file is *a.out*.**
- **Invokes a series of programs:**
 - preprocessor, compiler and linker
- **Compile sample.c program using the C compiler.**
 - enter the following command:
cc sample.c
 - Subprograms can be compiled together.



- 3 -

C Program development (CC , Make, SCCS)

An essential resource for C Program development is the C Compiler. A compiler is a special program that translates program source to executable code. If it encounters a error, it generates the error messages. The compiler also provides a suitable programming environment – file handling, basic I/O, other connections with the operating system.

In Program development there are source files, header files and library files. Library files are precompiled files which contain object code. The source and header files are created by programmers using an editor.(e.g. VI editor).

Source files in C have .c extension, c++ has .cpp extension, header files have .h extension.

Overview of Compilation and Linking

Compilation refers to the processing of source code files (.c,.cpp) and the creation of an 'object' file. This step doesn't create anything the user can actually run. Instead, the compiler merely produces the machine language instructions that correspond to the source code file that was compiled. For instance, if you compile (but do not link) three separate files, you will have three object files created as output, each with the name <filename>.o or <filename>.obj (the extension will depend on your compiler). Each of these files contains a translation of your source code file into a machine language file. But you cannot run them yet! You need to turn them into executables your operating system can use. That is where the linker comes in.

9.1: CC Command

Options with CC Command**➤ Specify your own output (.o) file name**

- By default cc creates output file by name a.out.
- To specify own file name use -o option
 - Example: `cc -o sample.o sample.c`
- CC compiler creates sample.o file instead of a.out
- Execute the file:
\$./sample.o --- "." represents current directory.

- 4 -



9.1: CC Command

Options with CC Command**➤ Some more options with cc**

- **-c:** Compiles only; does not attempt to link source files.
- **-l <Library>:** When linking, it adds the indicated library to the list of libraries to be linked.
 - C program includes mathematical functions such as *sin* or *tan*.
 - During compilation you must specify math library usage while linking.
 - Example: `cc sample.c -lm` --- Uses math library while linking.
- **-o output:** If linking, places executable output in the file output.

9.2: Make utility

Make Utility

- **Make utility and make files help automate application building.**
- **Modify any one or more source files.**
 - Requires *recompilation* or *re-linking* that part of the program.
 - Automate process
 - Specify interdependencies between files that make up the application.
 - Specify commands needed to recompile and relink each piece.



- 8 -

Managing the process of compiling large applications can be difficult. During program development a number of subroutines may be developed separately and linked at the end. However keeping a track of the updated versions is difficult. Further, files that are changed may not be recompiled. UNIX provides a tool that takes care of this for you. **make** looks for a makefile, which includes directions for building the application.


Make Utility

You can use the make utility and makefiles to help automate building of an application.

The make utility applies intelligence to the task of program compilation and linking.

Typically, a large application might exist as a set of source files and INCLUDE files, which require linking with a number of libraries. Modifying any one or more of the source files requires recompilation of that part of the program and relinking. You can automate this process by specifying the interdependencies between files that make up the application along with the commands needed to recompile and relink each piece. With these specifications in a file of directives, **make** ensures that only the files that need recompiling are recompiled and that relinking uses the options and libraries you want.

9.3 Running Make


iGATE
ITOPS for Business Outcomes

➤ Instructions can be placed in a file named **makefile**. Command **make** executes instructions defined in the **makefile**.

`$ make` -- Executes commands stored in a file named *makefile*

➤ **makefile** comprises a series of entries of the following form:

<code>[...targetfile...]:</code>	<code>[.....dependencies....]</code>
<code>[One Tab Space]</code>	<code>[.....commands.....]</code>


LEARN
Learning Education And Research Now

You can think of the makefile as being its own programming language. The syntax is as follows:

target: dependencies
Command list

Dependencies can be targets declared elsewhere in the makefile, and they can have their own dependencies. When a make command is issued, the target on the command line is checked; if no targets are specified on the command line, the first target listed in the file is checked.

9.3 Running Make

iGATE
ITOPS for Business Outcomes

➤ **Compile main.c through command prompt:**

cc main.c


➤ **Instead create a file with name *makefile* to compile main.c as follows:**

\$ vi makefile
main.o: main.c
cc -c main.c

To run makefile
\$make

➤ **Makefile and all files referenced by MAKE should be in the same directory.**

- 8 -

LEARN
Learning Education And Research Now

When make tries to build a target, first the dependencies list is checked. If any of them requires rebuilding, it is rebuilt. Then, the command list specified for the target itself is executed.

make has its own set of default rules, which are executed if no other rules are specified. One rule specifies that an object is created from a C source file using \$(cc) \$(CFLAGS) -c (source file). CFLAGS is a special variable; a list of flags that will be used with each compilation can be stored there. These flags can be specified in the makefile, on the make command line, or in an environment variable. make checks the dependencies to determine whether a file needs to be made. It uses the mtime field of a file's status. If the file has been modified more recently than the target, the target is remade.

9.3 An example of a makefile

Example

```

prog1 : file1.o file2.o file3.o
        cc -o prog1 file1.o file2.o file3.o
file1.o : file1.c mydefs.h
        cc -c file1.c
file2.o : file2.c mydefs.h
        cc -c file2.c
file3.o : file3.c
        cc -c file3.c
clean :
        rm file1.o file2.o file3.o

```



This is an example descriptor file to build an executable file called prog1. It requires the source files file1.c, file2.c, and file3.c. An include file, mydefs.h, is required by files file1.c and file2.c. If you wanted to compile this file from the command line using C the command is:

% CC -o prog1 file1.c file2.c file3.c

This command line is rather long to be entered many times as a program is developed and is prone to typing errors. A descriptor file could run the same command better by using the simple command:

% make prog1

or if prog1 is the first target defined in the descriptor file

% make

This first example descriptor file is much longer than necessary but is useful for describing what is going on.

Let's go through the example to see what make does by executing with the command make prog1 and assuming the program has never been compiled.

1. *Make* finds the target prog1 and sees that it depends on the object files file1.o file2.o file3.o
2. *Make* next looks to see if any of the three object files are listed as targets. So make looks at each target to see what it depends on. Make sees that file1.o depends on the files file1.c and mydefs.h.
3. Now *make* looks to see if either of these files are listed as targets and since they aren't it executes the commands given in file1.o's rule and compiles file1.c to get the object file.
4. *Make* looks at the targets file2.o and file3.o and compiles these object files in a similar fashion.
5. *Make* now has all the object files required to make prog1 and does so by executing the commands in its rule.

9.3 An example of a makefile

Example

- In above example, *clean* is a target which is not a file.
 - It is called as *phony* target.
 - Their common use is to remove files no longer needed after a program is made.

- 10 -



You probably noticed we did not use the target, *clean*, it is called a *phony target*.

A phony target is one that is not really the name of a file. It will only have a list of commands and no prerequisites. One common use of phony targets is for removing files that are no longer needed after a program has been made. The following example simply removes all object files found in the directory containing the descriptor file.

clean : rm *.o

9.3 Running Make

Continued...

➤ Table below displays results of executing make with these options.

Command	Result
<code>make</code>	use the default descriptor file, build the first target in the file
<code>make myprog</code>	use the default descriptor file, build the target <code>myprog</code>
<code>make -f mymakefile</code>	use the file <code>mymakefile</code> as the descriptor file, build the first target in the file
<code>make -f mymakefile myprog</code>	use the file <code>mymakefile</code> as the descriptor file, build the target <code>myprog</code>



- 11 -

\$make

It will use default descriptor file i.e. `makefile`

\$make myprog

It will use file named `makefile` and create target `myprog`.

In `makefile` if `myprog` is not the first target then you can specify it while executing `makefile`

\$make -f mymakefile

If your descriptor file name is different than the default name i.e. `makefile` then to execute your own file use `-f` option

In above example `make` will use `mymakefile` file instead of using `makefile`.

\$make -f mymakefile myprog

In above example `make` will use `mymakefile` instead of `makefile` to create target `myprog`

9.4: SVN - Subversion

What is SVN?**➤ Subversion (SVN):**

- Software versioning and revision control system.
- Tool that help in to control various program versions.
- Only incremental changes to the program are stored. Multiple copies are not stored.
- Particularly useful when programs are enhanced but the original version is still needed.



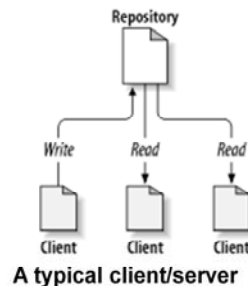
- 12 -

A version control system (or revision control system) is a system that tracks incremental versions (or revisions) of files and, in some cases, directories over time. Of course, merely tracking the various versions of a user's (or group of users') files and directories isn't very interesting in itself. What makes a version control system useful is the fact that it allows user to explore the changes which resulted in each of those versions and facilitates the arbitrary recall of the same.

9.4: SVN - Subversion

Repository

- **Repository is the central store of that system's data.**
- **The repository usually stores information in the form of a file system tree (a hierarchy of files and directories)**
- **Any number of clients connect to the repository, and then read or write to these files.**



- 13 -

At the core of the version control system is a repository, which is the central store of that system's data. The repository usually stores information in the form of a files system tree -- a hierarchy of files and directories. Any number of clients connect to the repository, and then read or write to these files. By writing data, a client makes the information available to others; by reading data, the client receives information from others.

What makes the repository special is that as the files in the repository are changed, the repository remembers each version of those files.

When a client reads data from the repository, it normally sees only the latest version of the files system tree. But what makes a version control client interesting is that it also has the ability to request previous states of the files system from the repository. A version control client can ask historical questions such as What did this directory contain last Wednesday? and Who was the last person to change this file, and what changes did he make? These are the sorts of questions that are at the heart of any version control system.

Note : Repository creation is a one time activity which happens on server and usually done by system administrator.

9.4: SVN - Subversion

Initial Project Setup

- If the project is not yet started , then create the basic directories required for project directly on repository.
- If the project is already in place and the directory structure has been already defined then import the directory structure (along with the all latest files) to the SVN repository.
- After importing now delete local copy as files are now successfully stored in SVN repository.
- Command for creating , Removing Directory and Checking log

```
svn mkdir file:///repository_path/directory_name -m "Message"  
svn rm file:///repository_path/directory_name -m "Message"  
svn log file:///repository_path
```



- 14 -

Note : Initial Project Setup creation is a one time activity which can be done by administrator or user .

Following is the command to import the entire project from client to server

svn import *user_directory_path* file:///repository_path -m 'Message'

9.4: SVN - Checkout

The Working Copy (Check-Out)

- An ordinary directory tree on local system, containing a collection of files is called working copy.
- Owner's private work area (edit , compile, change any files/code)
 - Complete Checkout

svn checkout file:///repository_path
 - Partial Checkout

svn checkout file:///repository_path/Directory_Path
- Changes in working copy needs explicit commit to update the changed version in repository

- 15 -



9.5: SVN – Lock, Modify, Unlock

The Problem of File-Sharing



➤ The challenge is :

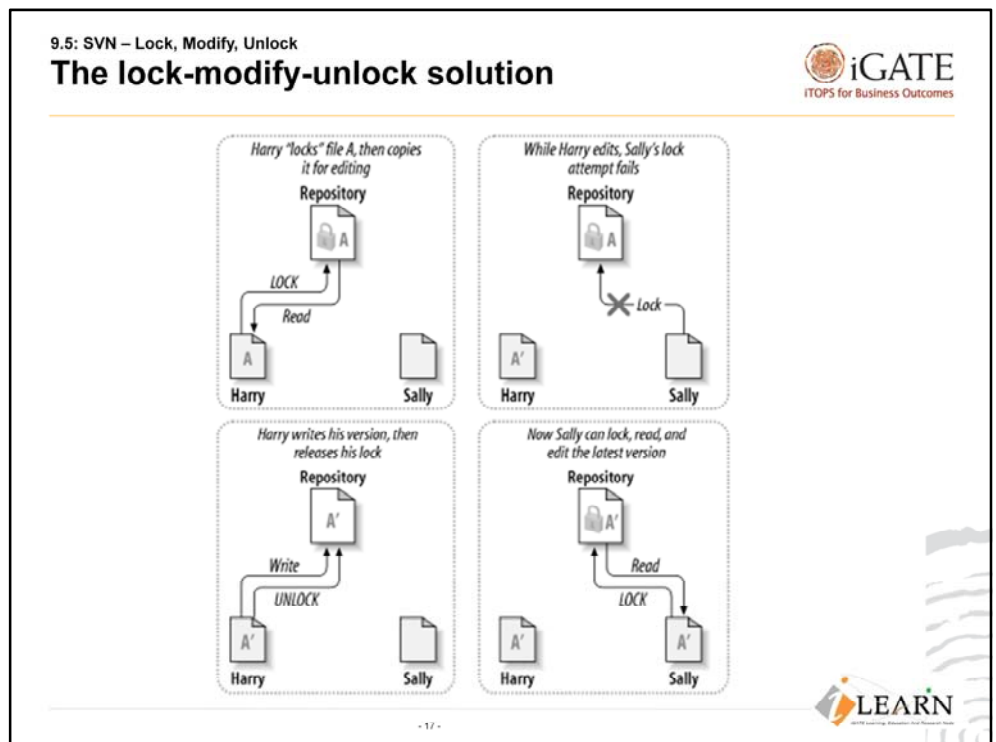
- How we can make sure that only one person at a time should modify the file



- 19 -

All version control systems have to solve the same fundamental problem: how will the system allow users to share information, but prevent them from accidentally stepping on each other's feet? It's all too easy for users to accidentally overwrite each other's changes in the repository.

Suppose we have two co-workers, Harry and Sally. They each decide to edit the same repository file at the same time. If Harry saves his changes to the repository first, then it's possible that (a few moments later) Sally could accidentally overwrite them with her own new version of the file. While Harry's version of the file won't be lost forever (because the system remembers every change), any changes Harry made won't be present in Sally's newer version of the file, because she never saw Harry's changes to begin with. Harry's work is still effectively lost or at least missing from the latest version of the file and probably by accident. This is definitely a situation we want to avoid!



The Lock-Modify-Unlock Solution

Many version control systems use a lock-modify-unlock model to address the problem of many authors clobbering each other's work. In this model, the repository allows only one person to change a file at a time. This exclusivity policy is managed using locks. Harry must lock a file before he can begin making changes to it. If Harry has locked a file, then Sally cannot also lock it, and therefore cannot make any changes to that file. All she can do is read the file, and wait for Harry to finish his changes and release his lock. After Harry unlocks the file, Sally can take her turn by locking and editing the file.

The problem with the lock-modify-unlock model is that it's a bit restrictive, and often becomes a roadblock for users:

- Locking may cause administrative problems. Sometimes Harry will lock a file and then forget about it. Meanwhile, because Sally is still waiting to edit the file, her hands are tied.

- Locking may cause unnecessary serialization. What if Harry is editing the beginning of a text file, and Sally simply wants to edit the end of the same file? These changes don't overlap at all. They could easily edit the file simultaneously.

The solution of this problem is "Copy-Modify-Merge"

9.6: SVN – Few More Commands

SVN-More Command➤ **Addition of file to repository**

```
svn add File_Path/File_Name File_Path/File_Name
```

➤ **Committing the changes**

```
svn commit -m 'Message'
```

➤ **Locking the file**

```
svn lock File_Path/File_Name File -m "Message"
```

➤ **Un-Locking the file**

```
svn unlock File_Path/File_Name
```

- 18 -



9.7: System Administration

Definition

- **Task to maintain the system.**
- **Task performed by Administrator:**
 - Maintain files.
 - Maintain user login ids.
 - Maintain system log, hardware, software and network administration.
 - Configure the kernel.
 - Install and patch the UNIX operating system.



9.7: System Administrator

Login as superuser

- A privileged user with *unrestricted* access to the whole system, all commands and all files regardless of their permissions.
- Username for the super user account is *root*.
- Two ways to login as root:
 - Directly login with username root.
 - Use su command.
- Programs a system administrator needs as root are kept in /etc e.g. etc/passwd.



+ 20 +

System Administrator

The superuser is a privileged user who has unrestricted access to the whole system; all commands and all files regardless of their permissions. By convention the username for the superuser account is root.

The root account is necessary as many system administration files and programs need to be kept separate from the executables available to non-privileged users. Unix allow users to set permissions on the files they own. A system administrator may need to override those permissions.

Because the superuser has the potential to affect the security of the entire system, it is recommended that this password be given only to people who absolutely need it, such as the system administrator. It is also a good idea to change the password on this account often.

There are several ways to log into the root account. If a system comes up in single user mode, whoever is logged in automatically has root privileges. When a system is already up in multi user mode, a user can log in directly as root. When a user is already logged in, issuing the su command, without options, will cause the system to prompt for the root password. Once it is given the user becomes root.


The root account has its own shell and frequently displays a prompt that is different from the normal user prompt. Commands and programs that a system administrator will need as root are kept in /etc to decrease the chances of a user trying them by accident. For example, /etc contains /etc/passwd, which holds a list of all users who have permission to use the system.

System Administrator (cont'd)

Because the root account has extensive privileges it has an equal potential for destruction. For example, the superuser may change another user's password without knowing the old password. The superuser can also mount and unmount file systems, remove any file or directory, and shut down the entire system. The root account should be used with caution and only when necessary to perform a given task. A misplaced keystroke in this mode can have disastrous results. Various Unix systems will have different utilities for system administration.

Super user can log in with the su command and can run utilities for system administration for creating, deleting and modifying groups and accounts.

9.8 Users & Groups

iGATE
ITOPS for Business Outcomes


Details

➤ **Users:**

- Each user on a system must have a login account
 - Unique username and UID number.
- The **/etc/passwd** file contains a list of users that the system recognizes.

➤ **Groups:**

- Each user in a system belongs to at least one group.
- Users may belong to multiple groups, up to eight or sixteen.
- List of all valid groups for a system are kept in **/etc/group**.

LEARN
Learning Experience and Research Network

- 21 -

Users

Each user on a system must have a login account identified by a unique username and UID number.

The **/etc/passwd** file contains a list of users that the system recognizes.

Groups

Each user in a system belongs to at least one group. Users may belong to multiple groups, up to a limit of eight or 16. A list of all valid groups for a system are kept in **/etc/group**.


This file contains entries such as:

work*:15:trsmith,pmayfiel,arushkin

Each entry consists of four fields separated by a colon. The first field holds the name of the group. The second field contains the encrypted group password and is frequently not used. The third field contains the GID (group ID) number. The fourth field holds a list of the usernames of group members separated by commas.


9.9 Passwd

Details



➤ **Passwords:**

- Stored in the system in *encrypted* form for security purpose.
 - Stored in **/etc/shadow** which is readable only by root.



- 22 -

Security Passwords

Unix deals with passwords by not storing the actual password anywhere on the system. When a password is set, what's stored is a value generated by taking the password that was typed in and using it to encrypt a block of zeros. When a user logs in, `/bin/login` takes the password the user types in and uses it to encrypt another block of zeros. This is then compared with the stored block of zeros. If the two match the user is permitted to log in.

Decryption of passwords is possible - this means that even encrypted passwords are not secure if kept in a world-readable file like `/etc/passwd`. `/etc/passwd` needs to be world-readable for a number of reasons, including the user's ability to change their own passwords. So passwords are stored in `/etc/shadow` which is readable only by root. Even this does not make passwords absolutely secure, it just makes them harder to get to.

9.10 Commands

Create new user➤ **Syntax**

```
useradd [-c comment] [-d home_dir]
        [-e expire_date] [-g initial_group] [-p passwd] login
```

➤ **Creates a new user account.**➤ **The *login* parameter must be a unique string.**➤ **Username cannot comprise ALL or default keywords.**

— Example:

```
$useradd myuser
```

- 23 -



The `useradd` command does not create password information for a user. It initializes the password field with an asterisk (*). Later, this field is set with the `passwd`

The `useradd` command always checks the target user registry to make sure the ID for the new account is unique to the target registry.

Different Options**-c *comment***

Supplies general information about the user specified by the *login* parameter. The *comment* parameter is a string with no embedded colon (:) characters and cannot end with the characters '#!'.

-d *dir*

Identifies the home directory of the user specified by the *login* parameter. The *dir* parameter is a full path name.

-e *expire*

Identifies the expiration date of the account. The *expire* parameter is a 10-character string in the *MMDDhhmm* form, where *MM* is the month, *DD* is the day, *hh* is the hour, *mm* is the minute, and *yy* is the last 2 digits of the years 1939 through 2038. All characters are numeric. If the *expire* parameter is 0, the account does not expire. The default is 0. See the **date** command for more information.

Useradd Command's Option**-g *group***

Identifies the user's primary group. The *group* parameter must contain a valid group name and cannot be a null value.

-G *group1,group2,...*

Identifies the groups the user belongs to. The *group1,group2,...* parameter is a comma-separated list of group names. with **-m** flag.

-m

Makes user's home directory if it does not exist. The default is not to make the home directory.

-r *role1,role2,...*

Lists the administrative roles for this user. The *role1,role2,...* parameter is a list of role names, separated by commas.

-s *shell*

Defines the program run for the user at session initiation. The *shell* parameter is a full path name.

-u *uid*

Specifies the user ID. The *uid* parameter is a unique integer string. Avoid changing this attribute so that system security will not be compromised.

9.10 Commands

Delete user account**➤ The *userdel* command**

- Removes the user account identified by the *login* parameter.
- Removes a user's attributes without removing their home directory by default.
- If **-r** flag is specified, then *userdel* command also removes the user's home directory.

- Example

- » Syntax: `userdel [-r] login`

```
userdel myuser
```



- 25 -

Userdel command

The *userdel* command removes the user account identified by the *login* parameter. The command removes a user's attributes without removing the user's home directory by default. The user name must already exist. If the **-r** flag is specified, the *userdel* command also removes the user's home directory.

9.10 Commands

Create, Change, Delete➤ **Create Group:**

```
$ groupadd <groupname>
```

```
Example: groupadd project1
```

➤ **Delete the Group:**

```
— groupdel <groupname>
```

```
Example : groupdel project1
```

➤ **Rename Group mygroup to project2:**

```
— $ groupmod -n project2 mygroup
```



- 25 -

If the group is not assigned during creation of the user account, the administrator can run this command to assign the new group to the new user account:

```
usermod -g <groupname> <username>
```

9.10 Commands

Change Passwd➤ **Change your own password.**`$passwd`➤ **Change password for user *myuser*.**`$passwd myuser`➤ **Delete passwd.**`$passwd -d myuser`

- 21 -



Summary



- CC
- MAKE
- SVN
- Introduction to system administrator



Review Questions



- _____ is the UNIX system compiler for C.
- **Modifying any one or more of the source files requires recompilation of that part of the program and relinking is possible because of make utility.**
 - True
 - False
- In _____ multiple copies are not stored only changes are stored
- List any 3 commands used in SVN?



Reference Books



- **The Unix Programming Environment**
 - Kerningham & Pike, Prentice Hall
- **Unix System V.4 Concepts and Applications**
 - Sumitabha Das, Tata McGraw-Hill
- **Advanced Unix Programmer's Guide**
 - Stephen Prata, BPB
- **Introducing Unix System V**
 - Vijay Mukhi, Tata McGrawHill

