Chapter 6

Tools

Chapter Objectives

To understand concepts of tools used over the duration of the course.

Objectives

For this chapter, the following are the objectives:

Overview of shell utilities
- sed; grep; tr; find;
- awk; perl

Overview of dev
- nm; ar; gcc; gdb; cvs; make,
- .so, .a library,
- writing man pages,
- rpm

Slide #6-1

Notes

In this chapter, we examine asynchronous events and Tools.

The objective of this chapter is to provide an understanding of concepts on Tools that are used over the duration of the course.

Chapter Organization

1. **Objective**: Introduction to
   - make,
   - .so, .a library files
   - gdb, gprof, eclipse, cvs,
   - writing man pages
   - rpm
2. **Description**: These are the various tools needed for systems programming
   This chapter provides an introduction to the concepts used in course.
3. **Concepts Covered in Chapter**:
   - Introduction to various tools.
   - LINUX Notes
4. **Prior Knowledge**:
   same as Chapter #1
5. **Teaching & Learning Strategy**:
   Discussion questions are,
   - What are these tools for?
6. **Teaching Format**:
   Theory + Homework Assignments
7. **Study Time**: 120 Minutes (Lecture & Theory)
   + ~45 minutes (Homework Assignments)
8. **Assessment**: Group Homework Assignments
9. **Homework Eval**: Group
10. **Chapter References**:

1. Unix Shell Commands & Utilities

sed

- “stream editor”
- various commands .. substitute data

Notes

1. sed reads a line into memory
   - 1.1. It then performs all the actions specified for it in memory, and replaces the line back in memory.
   - 1.2. It then reads the next line, and repeats.
   - 1.3. It does not change the content of data in the file. The output of sed will need to be written out to a new file.
2. By default, sed
   - 2.1. acts on the first pattern in the line. It can be restricted to act on a specific occurrence, or on all occurrences in the line.
   - 2.2. reads the entire file. It can be restricted to act on specific lines – either by range, or by pattern matching.
3. Examples
   - 3.1. change “temp” to “tmp”

```
echo "/temp/tmpfile" | sed 's/temp/tmp/'
```

   - 3.2. change “temp” to “tmp” and “tmpfile” to “myfile”

```
        echo "/temp/tmpfile" | \
sed ‘s/temp/tmp/’ | sed ‘s/tmpfile/myfile/’
```

alternatively,

```
echo “/temp/tmpfile” | \
      sed -e ‘s/temp/tmp/’ -e ‘s/tmpfile/myfile/’
```

3.3. deal with slashes in input

```
echo “/temp/tmpfile” | sed ‘s/\/temp/\/tmp/’
```

alternatively, use another character as the substitute command delimiter

```
echo “/temp/tmpfile” | sed ‘s~/temp~/tmp~’
```

3.4. Change all occurrences of a pattern

```
echo “/temp/tempfile” | sed ‘s/\/temp/\/tmp/g’
```

alternatively, use another character as the substitute command delimiter

```
echo “/temp/tempfile” | sed ‘s~/temp~/tmp~2
```

or

```
echo “/temp/tempfile” | sed ‘s/temp/tmp/2’
```

4. References

4.1. http://www.grymoire.com/Unix/Sed.html
4.2. http://www.gnulamp.com/sed.html

grep

- search patterns
- wildcard searches
- variant .. egrep allows expression-based searches.

1. grep too, reads a line into memory, and, searches for the pattern.

   1.1. egrep is a variant to grep .. allows expression based searches

2. Examples

   2.1. $ cat myfile.txt

```
This is line 1.
This is line 2.
This is lane 3.

This is also line 1.
This is not line 1.
```

   2.2. search "line" in file

```
$ grep line myfile.txt
    This is line 1.
    This is line 2.
    This is also line 1.
    This is not line 1.
```

   2.3. search "so" in file

```
$ grep so myfile.txt
```

```
        This is also line 1.
```

2.4. search all lines not containing “so” in file

```
$ grep –v “so” myfile.txt
        This is line 1.
        This is line 2.
        This is lane 3.

        This is not line 1.
```

2.5. search all lines not containing “so” and “not” in file

```
$ grep –v “so” myfile.txt
        This is line 1.
        This is line 2.
        This is lane 3.
```

2.6. search all lines not containing “so” and “not” in file

```
$ egrep –v “so|not” myfile.txt
        This is line 1.
        This is line 2.
        This is lane 3.
```

2.7. search all lines not containing “so” and “not” in file

```
$ egrep –v “l.ne” myfile.txt
        This is line 1.
        This is line 2.
        This is lane 3.
        This is also line 1.
        This is not line 1.
```

3. References

3.1. http://www.panix.com/~elflord/unix/grep.html
3.2. http://bulba.sdsu.edu/grephelp.html

tr

- variant of sed ..
- replace/remove one or more characters

Slide #6-2

Notes

1. tr can be used to replace or remove one or more characters.
2. Examples
   2.1. $ cat myfile.txt
   ```
   this is line 1.
   this is also line 1.
   this is not line 1.
   ```
   2.2. change a couple of characters
   ```
   $  tr is at < myfile.txt
   that at lane 1.
   that at alto lane 1.
   that at not lane 1.
   ```
   2.3. translate all to upper case
   ```
   $  tr [a-z] [A-Z] < myfile.txt
   THIS IS LINE 1.
   THIS IS ALSO LINE 1.
   THIS IS NOT LINE 1.
   ```
3. References
   3.1. http://www.ibm.com/developerworks/linux/library/l-tiptex5.html
   3.2. http://www.cyberciti.biz/faq/how-to-use-linux-unix-tr-command/

find

- search for files in a directory hierarchy
- find [path...] [expression]

xargs

- build and execute command lines from standard input
- typically used when many arguments

Slide #6-2

Notes

1. find can be used to search for files in a directory hierarchy.
2. Examples

2.1. List all files
```
$ find . -print
.
./11a.c
./11a.h
./myfile.txt
./11b.c
./11c.c
./11_Makefile
```
2.2. find files matching a pattern
```
$ find . -name "*a*" -print
./11a.c
./11a.h
./11_Makefile
```
2.3. find files matching a pattern
```
$ find . -name "*a*" -print
./11a.c
./11a.h
./11_Makefile
```
2.4. find files matching a pattern

```
$ touch file.a
$ find . -name "*a*" -print
./11a.c
./11a.h
./11_Makefile
./file.a
```

2.5. find files matching a pattern, that were modified in the last two days

```
$ find . -name "*a*" -mtime -2
./file.a
```

2.6. find files matching a pattern, that were unchanged in the last 21 days

```
$ find . -name "*a*" -mtime +21
./11a.c
./11a.h
./11_Makefile
```

2.7. find files matching a pattern, and containing a specific value

```
$ find . -name "*a*" -exec grep -i test {} \; -print
compile: testing
testing:
proc_write_test:
./11_Makefile
```

Notes

1. xargs can be used to process long list of arguments.
   Typically, xargs was used so that long lists of filenames could be handled one at a time, instead of running into a “Too many arguments error”.

2. Examples

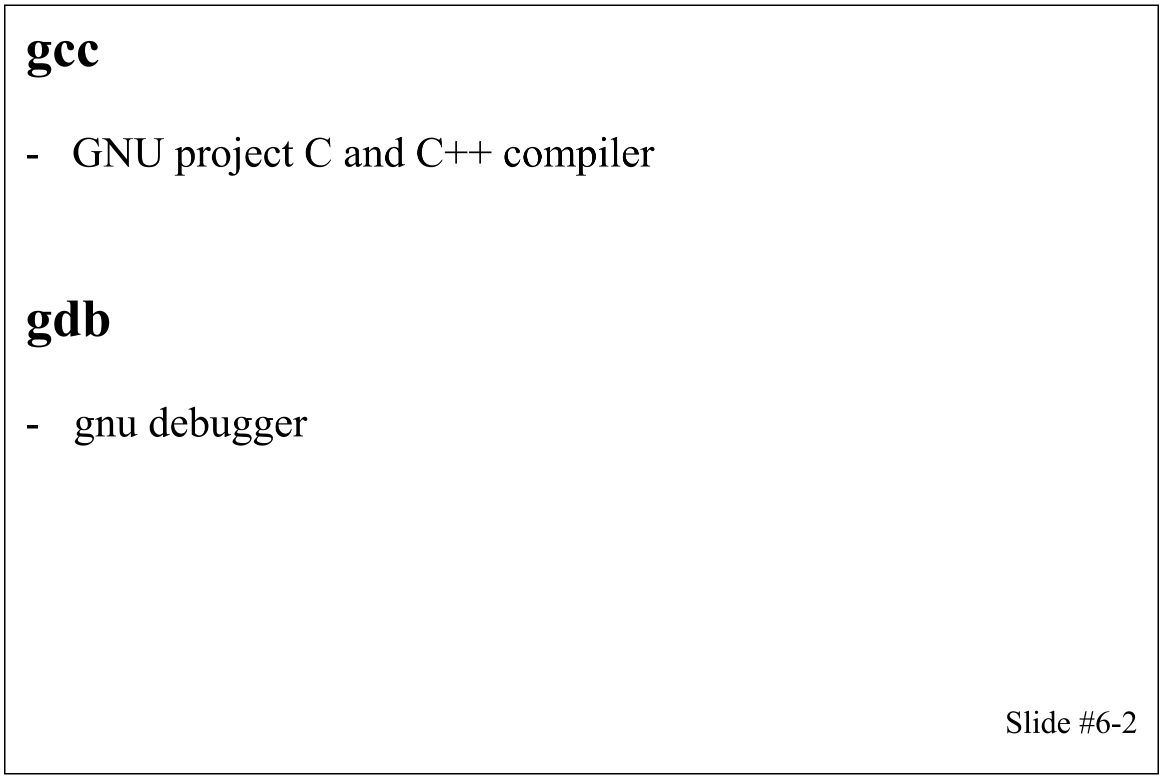
2.1. list all files

```
$  find . -name "*a*" -print | xargs ls
./11a.c  ./11a.h  ./11_Makefile  ./file.a
```

2.2. make a tar archive

```
$ find . -name "*a*" -print | xargs tar cvf test.tar
./11a.c
./11a.h
./11_Makefile
./file.a
```

sdk tools and utilities



gcc

- GNU project C and C++ compiler

gdb

- gnu debugger

Slide #6-2

Notes

1. C and C++ compiler.
   g++ accepts mostly the same options as gcc.

2. For the most part, the order of options and arguments doesn't matter. Order does matter when several options of the same kind are used; for example, if -L is specified more than once, the directories are searched in the order specified.

3. Examples

   3.1. sample program

```
$ cat myprog.c
#include <stdio.h>

int main() {
        printf("Hello World\n");
}
```

   3.2. Compile sample program, and get a.out

```
$ gcc myprog.c
$ ls a.out
a.out
```

   3.3. Compile sample program and rename a.out to myprog

```
$ gcc -o myprog myprog.c
$ ls myprog
myprog
```

3.4. Compile sample program and do not call linker

```
$ gcc –c myprog.c
$ ls myprog.o
myprog.o
```

Notes

1. GDB is invoked with the shell command gdb. It reads commands from the terminal until gdb exits.

2. gdb can be run with no arguments or options; but the most usual way to start GDB is with one argument or two, specifying an executable program as the argument:

   gdb program

3. Also, gdb can be started with both an executable program and a core file specified:

   gdb program core

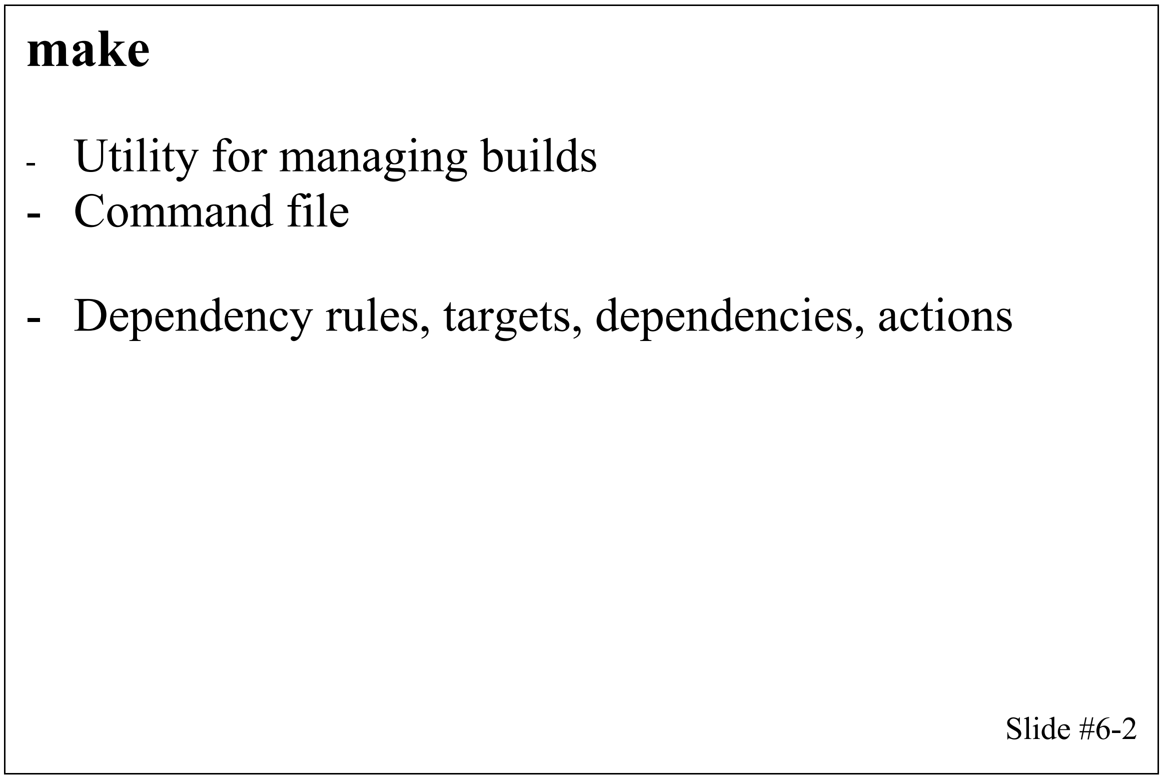
4. Also, a PID can be given as a second argument, for debugging a running process:

   gdb program 1234
   It would attach GDB to process 1234 (unless there is a file named "1234";
   GDB does check for a core file first).

5. Here are some of the most frequently needed GDB commands:

break [file:]function	Set a breakpoint at function (in file).
run [arglist]	Start your program (with arglist, if specified).
bt	Backtrace: display the program stack.
print expr	Display the value of an expression.
c	Continue running program (after stop e.g. at a breakpoint).
next	Execute next program line (after stopping); step over any function calls in the line.
edit [file:]function	look at the program line where it is presently stopped.
list [file:]function	type text of program in the vicinity where it is presently stopped.
step	Execute next program line (after stopping); step into any function calls in the line.
help [name]	Show information about GDB command name, or general information about using GDB.
quit	Exit from GDB.

make



make

- Utility for managing builds
- Command file
- Dependency rules, targets, dependencies, actions

Slide #6-2

Notes

make – Basic Concepts

1. Utility for managing builds.

   It provides a means to organize the compilation of source code.

   Compilation of multiple files from the command line is an error-prone, tedious and time consuming task. The “make” program helps manage this process and makes it much easier to deal with.

   ‘make’ looks for the modification timestamps of the program files, and, takes an action only if the timestamp of last modification is greater than the timestamp of the compiled target output. This is very useful when there are very small changes to a large program – only the files that got changed need to be compiled and linked into the final deliverable.

   Rules for compiling the project are contained in a file, usually called **makefile** or **Makefile**.

2. Components of makefile

   2.1. as earlier, each makefile has **rules** indicating how the target can be “made”

   Rules have the following syntax:
   ```
   output_files : input_files
         actions
   ```

   The first line of the rule is

- a space-separated list of output files
- followed by a colon,
- followed by a space-separated list of input files.

  The output files are also called *targets*, and input files are also called *dependencies*;

  The target depends on the dependencies, because if any dependency has changed, the entire target must be rebuilt.

The remaining lines of the rule are shell commands to be executed, and are known as actions.

- Each action must be indented with at least one tab character.
  Some make utilities need a space character.
- There can be as many action lines as are needed.
  Each line is executed sequentially.
  If any one of them fails, the remainder are not executed.
  The rule ends at the first line which is not indented.

Note: Each action line is executed as a separate sub-shell

2.2. In summary, rules have 3 parts : target, dependencies and actions
  - Dependency and Actions are optional.
  - Multiple Targets, Dependencies and Actions can be included into one rule.

3. The general syntax of a Makefile Target Rule is

```
target [target...] : [dependent ....]
    [ command ...]
```

Items in brackets are optional, ellipsis means one or more.
Note the tab to preface each command is required .. **mandatory**.

4. By default, make tries to make the first rule.

4.1. Make semantics are pretty simple.

4.2. At the shell command prompt, type "make target".

4.2.1. make finds the appropriate target rule and, if it finds that any of the dependents are newer than the target, the commands are executed one at a time - after macro variable substitution.

4.2.2. If any dependents have to be made, that happens first (so you have a recursion).

4.3. Each command in the target rule is executed in a separate shell.
Therefore, there can be fascinating make constructs and long continuation lines in the makefile.

5. Code Sample

```
all:    myprog1

myprog1:  myprog1.o
```

```
      $(CC) -o myprog myprog.o

myprog1.o: myprog1.c myprog.h
      ${CC} -o myprog1.o -c myprog1.c

myprog1.tar: myprog1.c myprog.h
      tar rf myprog1.tar myprog1.tar
```

6. General Rules of Syntax

6.1. Basic Make Rules

People expect certain targets in make files: make **all**, make **install**, make **clean**

make all	compile and make everything
make install	copy or install files in appropriate locations.
make clean	clean things up. It gets rid of objectfiles, executable files, and any temporary objects, files and directories etc.

6.2. **Comments**

Comments begin with the pound (#) sign. It can start anywhere on a line and continue until the end of the line. For example:

```
# This is a comment
```

6.3. **Macros**

6.3.1. Make supports variables that are also, called as "macros".

6.3.2. Make has a simple macro definition mechanism.
Macros are defined as Name=Value pairs.

6.3.3. To access the value for a macro, use braces or parenthesis

e.g. $(CC) or ${CC}

6.3.4. There are many default macros. To find them, use

make -p

```
AR = ar
GFLAGS =
GET = get
ASFLAGS =
MAS = mas
AS = as
FC = f77
```

```
CFLAGS =
CC = cc
LDFLAGS =
LD = ld
LFLAGS =
LEX = lex
YFLAGS =
YACC = yacc
LOADLIBS =
MAKE = make
MAKEARGS = 'SHELL=/bin/sh'
SHELL = /bin/sh
MAKEFLAGS = b
```

6.3.5. Environment Variables are exported into Make, as macros.
They override the defaults.

6.3.6. Special shortcut macro definitions are predefined.

$@	represents **target**. it may be denoted as `$(output)`
$<	represents **first dependency**. it may also be denoted as `$(input)`

- $< supports modifiers
- $(<F) represents **fileame** in first dependency.
- $(<D) represents **directory name** in first dependency.

$?	represents **changed dependents**.
$^	represents **all dependencies**. it may be denoted as `$(outputs)`
$*	represents **stem of pattern matching rule.** it may be denoted as `$(inputs)`

Code Sample

```
all:    myprog1


myprog1:  myprog1.o
     $(CC) -o $@ $^


myprog1.o: myprog1.c myprog.h
     ${CC} -o $@ -c $<


myprog1.tar: myprog1.c myprog.h
     tar rf myprog1.tar $?
```

6.4. **Continuation of Lines**

Uses a back slash (\).
This is important when the rule has to span across multiple lines .. for long macros and/or rules.

Pattern rules

A pattern rule is a concise way of specifying a rule for many files at once.
It is used when there are many source files.

It uses wildcards.

The "`%`" matches a string of any length, and stands for the string in the dependency list that matched.

The following pattern rule takes any `.c` file, and compiles it into a `.o` file:

```
%.o: %.c
 $(CC) $(CFLAGS) $(INCLUDES) -c $(input) -o $(output)
```

It assumes that variables `CC`, `CFLAGS`, and `INCLUDES` are defined

The first line of the rule is applied on every input file that matches the pattern `%.c`.
These `.c` files are built into corresponding `.o` files using following actions.

Actions in pattern rules use *automatic variables*- where the value is automatically set, depending on the rule that it appears in.

Automatic Variables

The most useful automatic variables are:

$(input)
$(output)
$(inputs)
$(outputs)

Note that these variables are lower case. Automatic variables can be used even in non-pattern rules to avoid repeating target filenames.

Also, it is easy to organize build environments using pattern rules and automatic variables.

```
# Put the object files into a separate directory:
OBJ/%.o: %.c
    $(CC) $(CFLAGS) -c $(input) -o $(output)
```

Pattern Rules and Automatic Variables

It is not uncommon for two different rules that produce the same file.

- If both are pattern rules, then the one that occurs later in the makefile is used.
- If one rule is a pattern rule, and the other is an explicit rule, then the explicit rule is used.

There is also another syntax that can affect compilation options for targets. A variable can have a different value for certain specific targets. In this example, it would look like this:

```
CFLAGS := -O2
DEBUG_CFLAGS := -g

%.o: %.c
 $(CC) $(CFLAGS) -c $(input) -o $(output)

debug.o: CFLAGS := $(DEBUG_CFLAG)
```

make has another syntax using the `:foreach` clause.

Builtin Rules

Make has builtin rules for compiling code. These can be overridden by user-specified builtin rules.

Phony Targets

The commands associated with make target do not actually build a file.

For example,

```
MYDIR=/home/appuser

install: myprog
    cp  $< $(prefix)/bin
    cp *.[ch] $(prefix)/sources

.PHONY: install
```

the line `.PHONY: install` .. denotes that the file `./install` will not exist after the actions are executed.

Without the phony declaration, "make" will expect the file `install` to exist after executing the commands.

The phony declaration can also be written as: $(phony install): myprog ...

And, the .PHONY: install line can be removed.

Calling Make across Directories .. recursively

If the sources are split into several directories, then dealing with several directories is much easier. Makepp is similar to standard unix make, however recursive invocations of make are easier. However, if the variable `$(MAKE)` is referenced in the makefile, makepp automatically goes into backward compatibility mode and turns off automatic loading.

With makepp, a separate makefile exists in each directory that builds the relevant files in that directory. When a makefile refers to files whose build commands are in different makefiles, makepp automatically finds the appropriate build rules in the other makefiles.

All actions in each makefile are executed with the current directory set to be the directory containing the makefile, so each makefile can be written independently of all the others.

No makefile has to know anything about the other makefiles; it does not even have to tell makepp to load the rules from those other makefiles.

Templates or boilerplate files

Information common to each makefile, such as variable definitions, and pattern rules can be put into a separate file.

```
include standard_defs.mk
```

The contents of the common file are inserted into the makefile at that point. The `include` statement first looks for the file in the current directory, then in the parent of the current directory, and so on up to the top level of the file system, so you don't actually need to specify actual path

7. VPATH environment variable.
   Can contain multiple directories, similar to PATH variable.
   Allows source files to be in a different directory.
   Targets and, object files will be created in current directory.

Code Sample

```
VPATH = $(HOME)/lsp/3743/src


myprog1.o:     myprog1.c myprog1.h
     $(CC) -o $@ $^
```

Libraries

nm

list symbols from object files

Slide #6-2

Notes

1. nm lists symbols from object files.

2. For each symbol, nm shows:
   - 2.1. The symbol value, in the radix selected by options (see below), or hexadecimal by default.
   - 2.2. The symbol type. At least the following types are used; others are, as well, depending on the object file format. If lowercase, the symbol is local; if uppercase, the symbol is global (external).

"A"	The symbol value is absolute, and will not be changed by further linking.
"B"	The symbol is in the uninitialized data section (known as BSS).
"C"	The symbol is common. Common symbols are uninitialized data. When linking, multiple common symbols may appear with the same name. If the symbol is defined anywhere, the common symbols are treated as undefined references.
"D"	The symbol is in the initialized data section.
"G"	The symbol is in an initialized data section for small objects. Some object file formats permit more efficient access to small data objects, such as a global int variable as opposed to a large global array.
“I"	The symbol is an indirect reference to another symbol. his is a GNU extension to the a.out object file format which is rarely used.
"N"	The symbol is a debugging symbol.

"R"	The symbol is in a read only data section.
"S"	The symbol is in an uninitialized data section for small objects.
"T"	The symbol is in the text (code) section.
"U"	The symbol is undefined.
"V"	The symbol is a weak object. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the weak symbol becomes zero with no error.
"W"	The symbol is a weak symbol that has not been specifically tagged as a weak object symbol. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the symbol is determined in a system-specific manner without error. Uppercase indicates that a default value has been specified.
"-"	The symbol is a stabs symbol in an a.out object file. In this case, the next values printed are the stabs other field, the stabs desc field, and the stab type. Stabs symbols are used to hold debugging information.
"?"	The symbol type is unknown, or object file format specific.

2.3. The symbol name.

3. Examples

3.1. list symbols .in a .o

```
$  nm myprog.o
 0000000000000000 T main
                  U printf
```

3.2. find a ".o" and, list the symbols in the .o

```
find . -name "*.o" -exec nm {} \; -print
0000000000000000 T main
                 U printf
./myprog.o
```

ar

create, modify, and extract from archives

Slide #6-2

1. ar is used to create, modify, and extract from archives
2. ar [-]p[mod [relpos] [count]] archive [member...]
3. GNU ar allows you to mix the operation code p and modifier flags mod in any order, within the first command-line argument. If you wish, you may begin the first command-line argument with a dash.

4. The p keyletter specifies what operation to execute; it may be any of the following, but you must specify only one of them:

   d Delete modules from the archive. Specify the names of modules to be deleted as member...; the archive is untouched if you specify no files to delete. If you specify the v modifier, ar lists each module as it is deleted.

   m Use this operation to move members in an archive. The ordering of members in an archive can make a difference in how programs are linked using the library, if a symbol is defined in more than one member. If no modifiers are used with "m", any members you name in the member arguments are moved to the end of the archive; you can use the a, b, or i modifiers to move them to a specified place instead.

   p Print the specified members of the archive, to the standard output file. If the v modifier is specified, show the member name before copying its contents to standard output. If you specify no member arguments, all the files in the archive are printed.

   q Quick append; Historically, add the files member... to the end of archive, without checking for replacement. The modifiers a, b, and i do not affect this operation; new members are always placed at the end of the archive. The modifier v makes ar list each file as it is appended. Since the point of this operation is speed, the archive's symbol table index is not updated, even if it already existed; you can use ar s or ranlib explicitly to update the symbol table index. However, too many different systems assume quick append rebuilds the index, so GNU ar implements q as a synonym for r.

   r Insert the files member... into archive (with replacement). This operation differs from q in that any previously existing members are deleted if their names match those being added. If one of the files named in member... does not exist, ar displays an error message, and leaves undisturbed any existing members of the archive matching that name. By default, new members are added at the end of the file; but you may use one of the modifiers a, b, or i to request placement relative to some existing member. The modifier v used with this operation elicits a line of output for each file inserted, along with one of the letters a or r to indicate whether the file was appended (no old member deleted) or replaced.

   t Display a table listing the contents of archive, or those of the files listed in member... that are present in the archive. Normally only the member name is shown; if you also want to see the modes (permissions), timestamp, owner, group, and size, you can request that by also specifying the v modifier. If you do not specify a member, all files in the archive are listed. If there is more than one file with the same name (say, fie) in an archive (say b.a), ar t b.a fie lists only the first instance; to see them all, you must ask for a complete listing---in our example, ar t b.a.

5. Modifiers A number of modifiers (mod) may immediately follow the p keyletter, to specify variations on an operationâs behavior:

a Add new files after an existing member of the archive. If you use the modifier a, the name of an existing archive member must be present as the relpos argument, before the archive specification.

b Add new files before an existing member of the archive. If you use the modifier b, the name of an existing archive member must be present as the relpos arg, before the archive specification. (same as i).

c Create the archive. The specified archive is always created if it did not exist, when you request an update. But a warning is issued unless you specify in advance that you expect to create it, by using this modifier.

f Truncate names in the archive. GNU ar will normally permit file names of any length. This will cause it to create archives which are not compatible with the native ar program on some systems. If this is a concern, the f modifier may be used to truncate file names when putting them in the archive.

i Insert new files before an existing member of the archive. If you use the modifier i, the name of an existing archive member must be present as the relpos argument, before the archive specification. (same as b).

l This modifier is accepted but not used.

N Uses count parameter. Used if there are multiple entries in archive with the same name. Extract or delete instance count of the given name from the archive.

o Preserve original dates of members when extracting them. If you do not specify this modifier, files extracted from the archive are stamped with the time of extraction.

P Use the full path name when matching names in the archive. GNU ar can not create an archive with a full path name (such archives are not POSIX compliant), but other archive creators can. This option will cause GNU ar to match file names using a complete path name, which can be convenient when extracting a single file from an archive created by another tool.

s Write an object-file index into the archive, or update an existing one, even if no other change is made to the archive. You may use this modifier flag either with any operation, or alone. Running ar s on an archive is equivalent to running ranlib on it.

S Do not generate an archive symbol table. This can speed up building a large library in several steps. The resulting archive can not be used with the linker. In order to build a symbol table, you must omit the S modifier on the last execution of ar, or you must run ranlib on the archive.

u Normally, ar r... inserts all files listed into the archive. If you would like to insert only those of the files you list that are newer than existing members of the same names, use this modifier. The u modifier is allowed only for the operation r (replace). In particular, the combination qu is not allowed, since checking the timestamps would lose any speed advantage from the operation q.

v This modifier requests the verbose version of an operation. Many operations display additional information, such as filenames processed, when the modifier v is appended.

V This modifier shows the version number of ar.

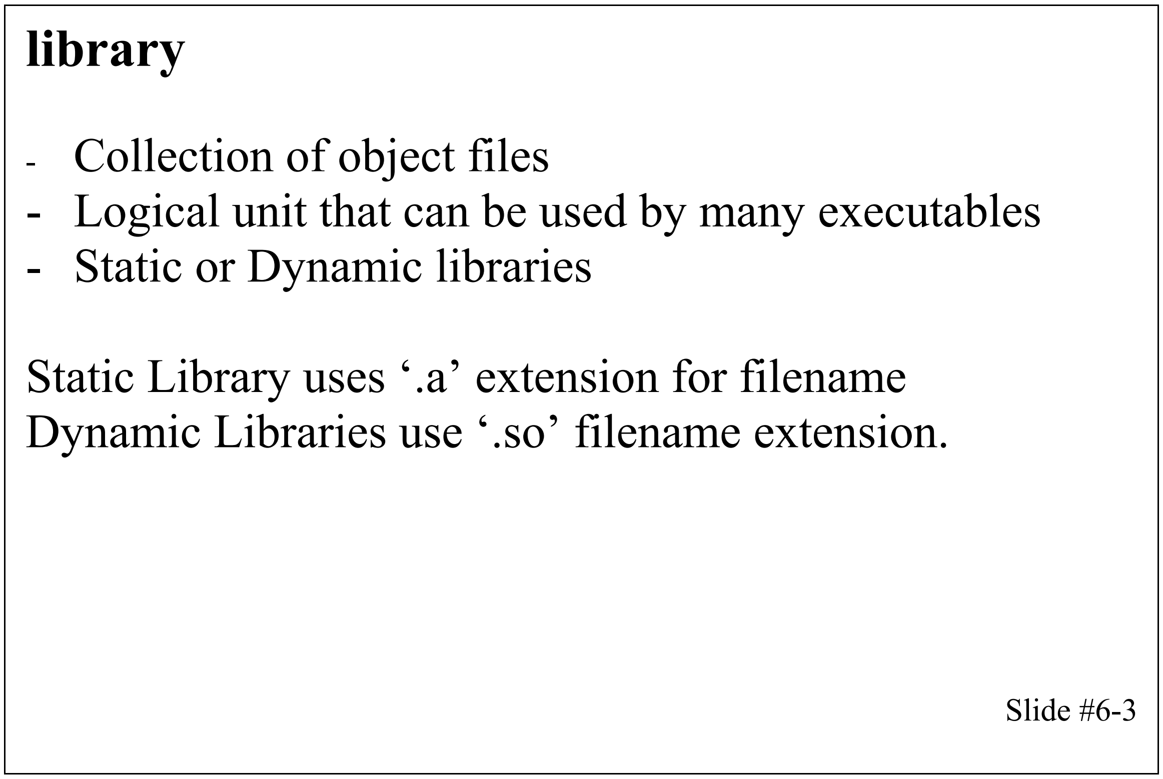
6. Examples

6.1. create an archive

```
$ ar cru libmylib.a myprog.o
$ ls libmylib.a
```

6.2. list contents of an archive

```
$ ar tv libmylib.a
rw-r--r-- 906/502   1496 Sep 24 12:41 2008 myprog.o
```



library

- Collection of object files
- Logical unit that can be used by many executables
- Static or Dynamic libraries

Static Library uses '.a' extension for filename
Dynamic Libraries use '.so' filename extension.

Slide #6-3

Notes

Basic Concepts – Static Library

1. to create an archive library use the 'ar' command
   ```
   ar cru libmylib.a myfn1.o myfn2.o myfn3.o
   ```
2. The next step is to run 'rannlib'
   ```
   ranlib libmylib.a
   ```
   2.1. adds a table of contents to the archive library, that converts the archive into a form, that can be linked rapidly.

   2.2. may or may not be needed.

      2.2.1. currently, on many systems, 'ar' generates the table of contents.

      2.2.2. since 'ar' generates the table of contents, 'ranlib' may be needed only to ensure **portable** scripts.

      2.2.3. on the other hand, some UNIX's like SGI Irix do not have the 'ranlib' command.

Basic Concepts – Dynamic or Shared Object Libraries

3. Compile using –fPIC flag
   ```
   gcc -fPIC -c myfn.c
   ```
   3.1. The –fPIC flag **may** not be needed on all compilers.
4. Build the dynamic library
   ```
   gcc -shared -Wl -o libmylib.so myfn.o
   ```

5. Specify LD_LIBRARY_PATH environment variable

```
$ export LD_LIBRARY_PATH=.:${LD_LIBRARY_PATH}
```

Basic Concepts – Next Steps

6. Once the library is created, it can be linked with other object files

   ```
   gcc myprog.o libmylib.a –o myprog
   ```

   6.1. the library can be installed into a standard directory – usually '/usr/lib' and '/usr/local/lib'

   6.1.1. copy the library into the standard directory, so that

   6.1.2. the linker can find the library

   6.2. the compiler can be directed to use the new library

   ```
   gcc –o myprog –lmylib
   ```

   6.2.1. –lmylib .. includes prefix 'lib' and the suffix '.a' or '.so.' Assumes library can be found in standard directory.

   6.2.2. if the library is not in the standard directory, the '-L' compiler directive specifies the location.

   ```
   gcc –o myprog –L mydir –lmylib
   ```

   - The '–L' must appear before the –l flag.
   - Using multiple –L flags, is not uncommon.
   - The multiple '-L' flags can be in any order, so long as they precede the –l specification.
   - When using multiple –l flags, the order in which the libraries are linked is important. The linker does not embed the whole library into the executable, but only symbols as needed by the executable.
   - If there are dependent libraries, then higher-level libraries (those used by the object file) would need to be specified prior to the lower-level libraries.

7. 'ldd' can be used to list the dynamic dependencies of executables, or shared objects.

```
ldd /bin/ls
     libc.so.1 =>     /usr/lib/libc.so.1
     libdl.so.1 =>    /usr/lib/libdl.so.1
     /usr/platform/SUNW,Ultra-4/lib/libc_psr.so.1
```

8. **code example**

```
/*  myprog.c */

#include <stdio.h>

extern int myfn(void);
```

```
int main() {

        myfn();

}
```

```
/* myfn.c */

#include <stdio.h>

int myfn () {
        printf("Hello World\n");
}
```

Compiler and Sample commands – Static Library

```
$ gcc -c myfn.c
$ ar cru libmylib.a myfn.o
$ ranlib libmylib.a

$ gcc -c myprog.c
$ gcc -o myprog myprog.o -L. -lmylib

$ ./myprog
Hello World
```

Compiler and Sample commands – Dynamic Library

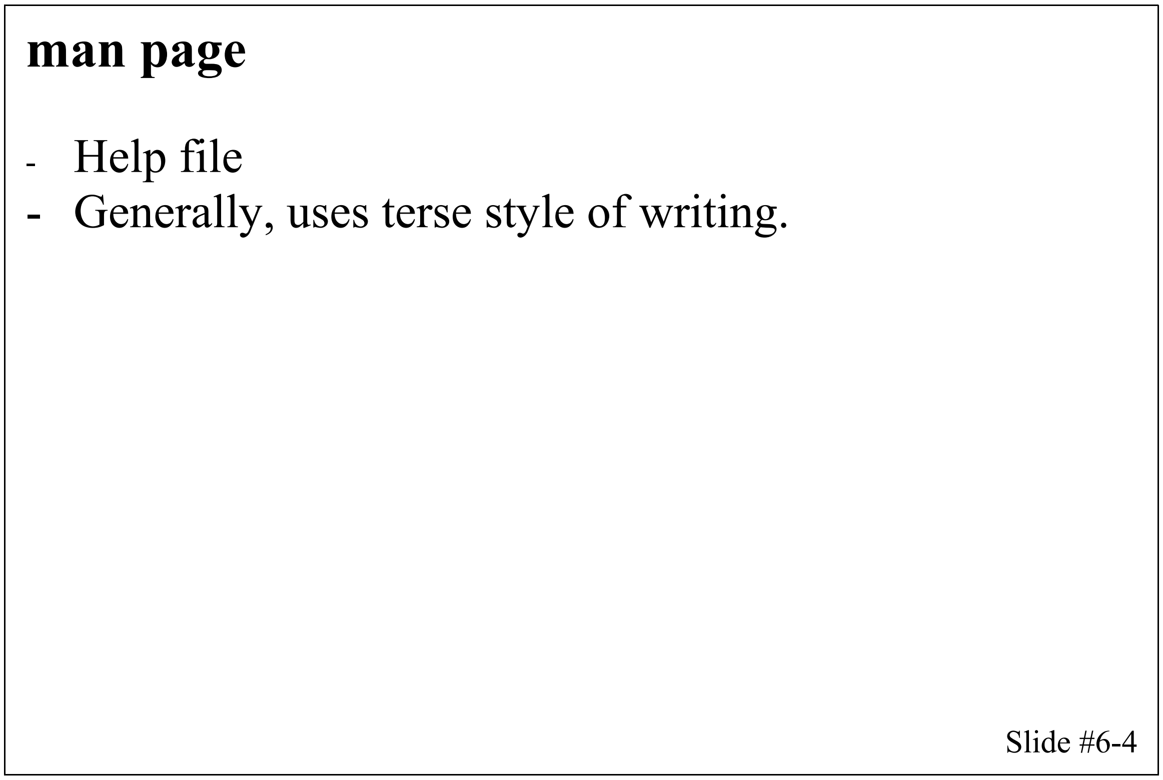
```
$ gcc -fPIC -c myfn.c
$ gcc -shared -Wl -o libmylib.so myfn.o

$ gcc -c myprog.c
$ gcc -o myprog myprog.o -L. -lmylib

$ export LD_LIBRARY_PATH=.:${LD_LIBRARY_PATH}
$ ldd myprog
        libmylib.so =>   ./libmylib.so
        libc.so.1 =>     /usr/lib/libc.so.1
        libdl.so.1 =>    /usr/lib/libdl.so.1
        /usr/platform/SUNW,Ultra-4/lib/libc_psr.so.1

$ ./myprog
Hello World
```

man pages



man page

- Help file
- Generally, uses terse style of writing.

Slide #6-4

Notes

Basic Concepts – man page

1. manual pages –
   1.1. provide usage information on syntax and command options.
   1.2. generally, uses terse style of writing.
2. organized in a 'fixed' format.
   Sections include

NAME	- name and short description of program
SYNOPSIS	- command line syntax format
DESCRIPTION	- describe program,
OPTIONS	- explanation of command line options
BUGS	- list of known bugs
AUTHOR	- author name and email address
SEE ALSO	- any other related commands and programs.

Basic Concepts – man

3. man command runs 'groff' command.   (or, 'roff', 'nroff' command as applicable)
   3.1. groff is a text formatter.
   It reads special macros in a file and outputs a formatted file.

3.2. groff macros always start with a '.', followed by macro name and parameters.

3.2.1.	.TH	- Title Header
3.2.2.	.SH	- Section Header

Optionally can use

3.2.3.	.B	- bold
3.2.4.	.I	- Italics
3.2.5.	.R	- Roman

can combine above, into

	.BI	- alternating bold italics, similarly .BR, .IB, .IR, .RB, .RI
3.2.6.	.\"	- comments
3.2.7.	.PP	- Paragraphing
3.2.8.	.RS,. .RE	- Indentation Start and End. RS = Relative Indent Start; RE = Relative Indent End.
3.2.9.	.IP	- Indent Paragraph, Typically, used with options takes an argument. Anything else on other lines is tabbed, and indented right.
3.2.10.	.TP	- Tag Paragraph

Basic Concepts – Creating a man page

4. .TH macro

.TH [name] [section] [center footer] [left footer] [center header]

4.1. TITLE HEADER
expects 5 parameters as above.

4.2. name - name of program, that will also be in the top header, on the right.

4.3. section - man page section

man pages are organized into sections

4.3.1.	Section 1	user commands.
4.3.2.	Section 2	system calls.
4.3.3.	Section 3	subroutines.
4.3.4.	Section 4	devices.
4.3.5.	Section 5	file formats.
4.3.6.	Section 6	games.
4.3.7.	Section 7	miscellaneous.
4.3.8.	Section 8	system administration

4.4. center footer	displayed on every page. Normally, date.
4.5. left footer	displayed on every page. Normally, program or OS version number.
4.6. center header	displayed on every page. Normally, omitted.

5. Generally, any white space needs to be quoted.
   A'-' may need to be prefixed with a '\' i.e. use '\-'
6. to test use,
   groff -man -Tascii ./myprog.1 | less
   or

   man ./myprog.1

Basic Concepts – Installing a man page

7. The following are the steps:
   7.1. Compress with gzip
   7.2. Copy to respective man pages directory.
   7.3. Run 'makewhatis' to add to whatis database

```
# gzip myprog.1
# cp myprog.1.gz /usr/man/man1
# makewhatis
# whatis myprog
# man myprog
```

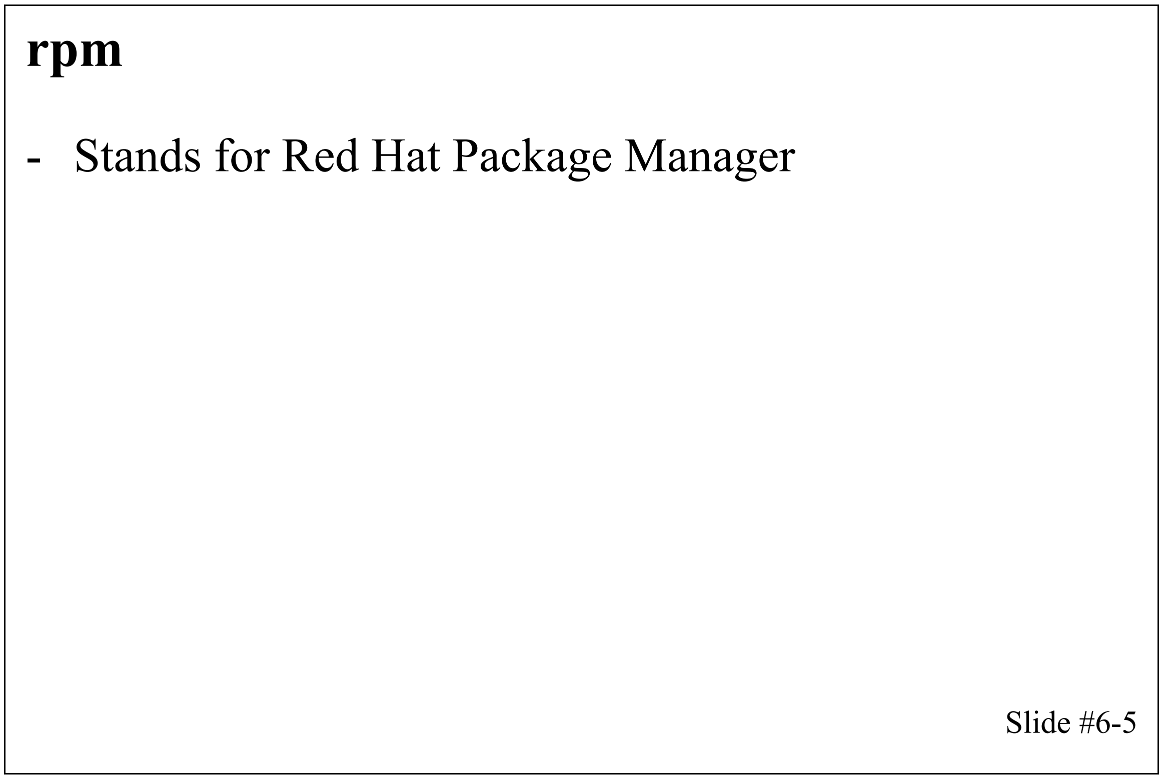
8. Code Sample:

```
.\" Process this file with
.\" groff -man -Tascii foo.1
.\"
.TH FOO 1 "MARCH 2005" Linux "User Manuals"
.SH NAME
mysh \- mysh my very own first shell program
.SH SYNOPSIS
.B mysh [-h] [-c
.I command-file
.B ]
.I file
.B ...
.SH DESCRIPTION
.B mysh
is the shell or command language interpreter for Linux.  The name
.BR 'mysh' (1)
represents the first shell program as written by me.
.SH OPTIONS
.IP -h
display help on options.
.IP "-c command-file"
accept commands in
.I config-file
```

```
to execute. Alternatively,
.IR ~/.myshrc
can be used. It overrides any
.B MYSHCONF
environment variable.
.SH FILES
.I ~/.myshrc
.RS
The user specific command file. See
.BR .myshrc (5)
for further details.
.RE
.SH ENVIRONMENT
.IP MYSHCONF
If it is non-null, then the full pathname for an alternate command file
.IR ~/.myshrc
.SH DIAGNOSTICS
The following diagnostics may be issued on stderr:

Bad Command or Filename.
.RS
The Command was not recognized.
.RE
Invalid Input.
.RS
.B mysh
can only a limited-set of commands. Try using a different shell.
.SH BUGS
This is an experimental shell.  Use at your own risk.
.SH AUTHOR
Linux Systems Programming <lsp at mysh dot net>
.SH "SEE ALSO"
.BR mysh2 (1),
.BR .myshrc (5),
.BR mysh3 (1)
```

rpm



rpm

- Stands for Red Hat Package Manager

Slide #6-5

Notes

Basic Concepts – rpm

1. package –
   - 1.1. simplifies the task of installing new software.
   - 1.2. encapsulates different files – man pages, source code, binaries, compilation commands.

Basic Concepts – Creating rpm

2. Creating an RPM involves 5 basic steps
   - 2.1. Create an RPM build environment.

```
$ mkdir -p ~/RPM/tmp ~/RPM/BUILD ~/RPM/RPM/i386 ~/RPM/RPMS/i686
~/RPM/RPMS/noarch ~/RPM/SOURCES ~/RPM/SPECS ~/RPM/SRPMS

$ echo “%_topdir        ~/RPM”       >> ~/.rpmmacros

$ echo “%_tmppath       ~/RPM/tmp”   >> ~/.rpmmacros
```

   - 2.2. Install rpm-build software http://www.rpmfind.net/linux/rpm2html/search.php?query=rpm-build
   - 2.3. Customizing rpm-build and modifying the install routines.
     This step is to modify the process by which the software is normally built and installed. The Makefile or install script is modified so that, when **rpm**-build is

building the software, it knows to place the files in a subdirectory of the build root (ie. that temporary directory where the files will be placed).

If the Makefile or automake file is well written, there is no modification needed as rpmbuild relies on many standard names. It is sometimes easier to simply install the files manually from inside the spec file into the appropriate directories for rpmbuild. If changes are needed. rpmbuild makes this easy. When installing, there are a wide variety of variables to indicate where things should go.

Examples:

- $RPM_BUILD_ROOT - where files will go. Generally different for every **RPM** spec file. Typically something like ~/**rpm**/tmp/nameofpackage-root, assuming you created ~/.rpmmacros as described above.
- $RPM_DOC_DIR - standard place for documentation files from RPMs. Perhaps /usr/share/doc.
- %{_bindir} (or $bindir inside the Makefile) - this variable will be defined as the standard place to put binaries, relative to build root. Something like ${RPM_BUILD_ROOT}/usr/bin.
- %{_libdir} - similar to bindir.

There are many more. Makefile's should expect and use these variables when deciding where to install things. For example, at the beginning of a Makefile's, the following lines can be added:

```
# These are typically taken from rpm, but, if not,
defined here.
bindir=/usr/local/bin
libdir=/usr/local/lib
sysconfdir=/etc
mandir=/usr/local/man
ifdef RPM_DOC_DIR
myshdir = $(RPM_BUILD_ROOT)/$(RPM_DOC_DIR)/$
(RPM_PACKAGE_NAME)-$(RPM_PACKAGE_VERSION)
else
myshdir = /usr/local/doc/mysh
endif
```

The first few lines define default install locations. If rpmbuild is building the software, these will be ignored and the values rpmbuild provides will be used. The ifdef part looks for a variable which indicates if rpmbuild is present and uses it if so.

2.4. Writing a spec file.

See http://www.ibm.com/developerworks/library/l-rpm1/

2.5. Building the RPM

Finally to make an RPM.
$ cp mysh.spec ~/**rpm**/SPECS/

```
$ tar -zcvf ~/rpm/SOURCES/mysh-myversion.tar.gz mysh-myversionnumber
$ rpmbuild -ba ~/rpm/SPECS/mysh.spec
```

Notice that source files should be in a directory with the appropriate name and version number before tar'ing.

2.6. Building the RPM in the home directory, without root access

2.6.1. The RPM can be built in the home directory.

2.6.2. create a file in the home directory, and call it .rpmmacros.

2.6.3. The contents of the .rpmmacros is:

- replace "raghav" with the correct homedir information.

```
%_topdir /home/raghav/rpm
%_tmppath /home/raghav/rpm/tmp
%_signature gpg
%_gpg_name Raghav V
%_gpg_path ~/.gnupg
%distribution RedHat Linux 9
%vendor RedHat
```

2.6.4. Then, create a file called .rpmrc in the same home directory, and it contains

```
buildarchtranslate: i386: i386
buildarchtranslate: i486: i386
buildarchtranslate: i586: i386
buildarchtranslate: i686: i386
buildarchtranslate: athlon: i386
```

2.6.5. Finally, create the directory structure required for doing rpm builds:

```
mkdir ~/mkdir ~/rpm/BUILD
mkdir ~/rpm/RPMS
mkdir ~/rpm/RPMS/athlon
mkdir ~/rpm/RPMS/i386
mkdir ~/rpm/RPMS/i486
mkdir ~/rpm/RPMS/i586
mkdir ~/rpm/RPMS/i686
mkdir ~/rpm/RPMS/noarch
mkdir ~/rpm/SOURCES
mkdir ~/rpm/SPECS
mkdir ~/rpm/SRPMS
mkdir ~/rpm/tmp
```

2.6.6. Rpm build will now use the structure in the home directory instead of the /usr/src/redhat directory.

Chapter 6

Tools

Assignment Questions

Questions:

6.1 Modify "mysh" (from chapter 5) to include commands

- 6.1.1 'mywc', 'mycat' and 'myls' from previous chapter(s), as built-in commands,
- 6.1.2 Each is in a different file (.c and .o)
- 6.1.3 Create a static or dynamic library for these commands.

6.2 Create a man page for 'mysh' – stating the builtin commands. Test it.

6.3 Create a makefile to re-create lib archive, or 'mysh' – as appropriate. Test it.

6.4 Create an rpm for 'mysh'. Test it.

6.5 Modify your makefile to add a make rule for creating an rpm for 'mysh'. Test it.

Assignment Hints

http://www.redhat.com/docs/books/max-rpm/max-rpm-html/index.html

Useful Links

Man pages

http://www.justlinux.com/nhf/Miscellaneous/Creating_Your_Man_Page.html

http://www.unixreview.com/documents/s=8925/ur0312i/

http://tldp.org/HOWTO/Man-Page/index.html

http://www.schweikhardt.net/man_page_howto.html

--

rpm

http://en.wikipedia.org/wiki/RPM_Package_Manager

https://pmc.ucsc.edu/~dmk/notes/RPMs/Creating_RPMs.html

http://www.ibm.com/developerworks/library/l-rpm1/

http://genetikayos.com/code/repos/rpm-tutorial/trunk/rpm-tutorial.html

http://docs.python.org/dist/creating-rpms.html

--

http://www.amath.washington.edu/~lf/tutorials/autoconf/toolsmanual.html#SEC21

http://www.mtsu.edu/~csdept/FacilitiesAndResources/make.htm

http://www.cs.colorado.edu/~kena/classes/3308/f04/lectures/

http://makepp.sourceforge.net/1.19/makepp_tutorial.html

http://www.metalshell.com/view/tutorial/120/

--

automake

--

autoconf

--

http://en.wikipedia.org/wiki/GDB

http://sourceware.org/gdb/

http://www.gnu.org/software/ddd/

--

gprof

http://sourceware.org/binutils/docs-2.17/gprof/index.html

http://sam.zoy.org/writings/programming/gprof.html

--

rpm

http://www.ibm.com/developerworks/library/l-rpm1/

--

Nm command

http://linux.about.com/library/cmd/blcmdl1_nm.htm

http://www.linux.com/base/ldp/howto/Program-Library-HOWTO/miscellaneous.html

--

Linker Info

http://www.securityfocus.com/infocus/1872

--

General debugging Techniques

http://publib.boulder.ibm.com/infocenter/javasdk/v1r4m2/index.jsp?topic=/com.ibm.java.doc.diagnostics.142j9/html/toolslinux.html