$Id: lab2u-utilities-c-make.mm,v 1.68 2015-01-16 12:06:23-08 - - $
PWD: /afs/cats.ucsc.edu/courses/cmps012b-wm/Labs-cmps012m/lab2u-utilities-c-make
URL: http://www2.ucsc.edu/courses/cmps012b-wm/:/Labs-cmps012m/lab2u-utilities-c-make/

1. Overview

In this lab you will continue being familiarized with Unix, looking at some utilities: **find**(1), **grep**(1), **man**(1), **du**(1), etc. We will also introduce the C language and the **make** utility.

2. The man(1) command

The **man**(1) command can be used to read about any Unix command, C language function, or other Unix utility. When referenceing a man-page, following the name is a number in parentheses which identifies the chapter in question. Following are the sections and what they describe:

(1) General commands
(2) System calls
(3) Library functions, covering in particular the C standard library
(4) Special files (usually devices found in **/dev**) and drivers
(5) File formats and conventions
(6) Games and screensavers
(7) Miscellanea
(8) System administration commands and daemons

3. The find(1) and grep(1) commands

The **find**(1) command can be used to find files in directory hierarchies according to various options. The general syntax is

find [*path* . . .] [*operand* . . .]

The pathnames specified identify one or more directories. There are many operands, some of which are:

-name *wildcard*	Finds files according to the standard shell wildcard expressions, with case being significant.
-iname *wildcard*	Finds files according to the standard shell wildcard expressions, with case not being significant.
-mtime *days*	Finds files according to the time stamp, measuread in days. For example **+3** means more than 3 days, **-3** means less than 3 days, and **3** means exactly 3 days.
-size *units*	Finds files according to their size. For example if the units are specified as **+200k**, any files larger than that will be found.

The **grep**(1) command is used to search a sequence of files, given a regular expression. As long as metacharaters are not used, it can be used to search for strings. Regular expressions are somewhat more complicated.

4. Pipes

A pipe is a connection between the output of one process and the input of another process and can be used to connect processes together into a pipeline. For example, **ls**(1) lists information about files, and **wc**(1) prints the number of lines, words, and characters in its input. Therefore:

```
bash-01$ ls | wc -l
```

will count the number of (non-dot) files in your current directory.

```
bash-02$ ls -la | grep 'Jan 13'
```

will list the output of `ls` for all files modified on January 13.

5. Lab exercises

Each of the following items will require something to be submitted for credit in this lab. It is assumed that the previous lab has been completed.

(1) Read the man page for **find**(1):

```
bash-03$ man -s 1 find
```

Then redirect the output into a file called `man.1.find`, and note that if you view it with an editor, you will see a lot of backspace characters represented as "`^H`" characters. On a text terminal, this would overstrike to make some characters bold. Clean this up by using:

```
bash-04$ vim man.1.find
```

Then use the **vim** line-mode command

```
:g/.^V^H/s///g
```

which will clean it up and make it readable as text. Note that "`^V^H`" means type "`vh`" while holding down the Control key.

Submit: `man.1.find`

(2) Repeat this exercise for the **grep**(1) command.

Submit: `man.1.grep`

(3) Use the **find** command to locate all PDF files in my `Assignments/` and `Labs-cmps012m/` directories.

```
bash-05$ find ~/12b/[AL]* -name '*.pdf'
```

Note that the first few operands are directories (expanded via wildcards) which specify directories, and the `-name` operand specifies wildcards for the names of files to match. Redirect this into a file called `asgs-labs-found`

Submit: `asgs-labs-found`

(4) Use the **grep**(1) command to search for the string "`Submit:`" in all of the lab directories for all files with the suffix `.tt`:

```
bash-06$ grep Submit: lab*/*.tt
```

Note that you have to **cd** into the directory `Labs-cmps012m/` for this to work. Redirect the output of this command into a file called `grepped-submits`. Note that you can not create this file in the course directory, so you will have to specify a pathname which deposits this file in one of your directories.

Submit: `grepped-submits`

(5) The **du**(1) (disk usage) command performs a recursive traversal over all directories and prints out the amount of disk space taken by each directory. This is useful in finding out if you are using up too much disk space. The command **fs lq** lists your disk quota.

```
bash-07$ fs lq ~
bash-08$ du -k ~/private/cmps012b
```

Print out your current disk quota and disk usage for your 12B and 12M directories. (If you have named your 12B and 12M working directories something else, use the actual name if different from above.) The **-k** option specifies that disk usage should be measured in K-bytes ($1K = 2^{10} = 1024$). Redirect the output of both of these commands into a file called **disk-usage**.

Submit: **disk-usage**

Note that to append to the end of a file you use **>>** instead of **>**. For example, the following will create a file called **foo** with your current working directory followed by the date:

```
bash-09$ pwd >foo
bash-10$ date >>foo
```

6. Introduction to C

Copy the program **code/hello.c** into your directory. Modify the program to conform to the specifications of **hello.java** in the previous lab.

(1) The program argument **argc** counts the number of command line arguments given to the program.

(2) The program argument **argv** is a pointer to an array of pointers to character string, where **argv[0]** is the name of the program itself.

(3) At the top of **main**, add an **int** variable called **exit_status** which is initialized to the value **EXIT_SUCCESS**.

(4) If there are no command line arguments, **argc** should have the value **1**. If that is the case, print the message

```
Hello, World!
```

as is done in the starter code.

(5) If not, print the message

```
Usage: hello
```

to **stderr**. **fprintf**(3) works like **printf**(1), except that its first argument should be **stderr**, followed by the format.

(6) The program name should not be hard-coded. instead use the expression

```
basename (argv[0])
```

which obtains the name of the program without the directory information.

(7) In the case of failure, set **exit_status** to **EXIT_FAILURE**, and return that value as the value of the program.

Submit: **hello.c**

7. Introduction to make

A **Makefile** is used to build software from specifications. Copy **code/Makefile** into your lab2 project directory and rename it to **Makefile**. (Do not copy the file **Makefile** in this directory. It builds the PDF from the groff sources.)

(1) The first line contains an RCS **Id** header in a comment, which starts with **#**, not a double slash.

(2) Then some variables are defined, with `OBJECTS` being the same as `SOURCES`, except for the suffix.

(3) Note the options given to `gcc`, which is used to compile C code.

(4) Then there are dependencies which compile the source into object files, and link the object files into an executable binary.

(5) By analogy we might consider object files to be similar to class files, and the executable image similar to a jar. Analogies only work so far then fail.

(6) Add a target `test` to the make file and put commands after it. Each command must be indented by one tab. The test should do the following :

   (i) Run `hello` and redirect its output to `test1.out`, its error to `test1.err`, then ***on the same line***, redirect its exit status to `test1.status`. Note that the `hello` command and the `echo` command must be on the same line separated by a semi-colon.

   (ii) Run `hello` with an argument of `world` and redirect its `stdout` and `stderr` as before but call the files `test2.out`, `test2.err`, and `test2.status`.

(7) In a `Makefile`, the dollar sign always introduces a variable whose value is substituted. This can be done either with a single character variable name or multiple characters in braces. So, for example, `$< $@` and `${FOOBAR}` are all variables to be substituted. So `$?` substitues the value of the variable `?`, which, unless defined, is replaced by nothing. The variable `$$` has the value of '`$`', so to send `$?` to the shell, write `$$?` in the `Makefile`.

Submit : `Makefile`

```c
// $Id: hello.c,v 1.1 2015-01-13 15:48:01-08 - - $

//
// NAME
//    hello - print the "Hello, World!" message.
//
// SYNOPSIS
//    hello
//
// DESCRIPTION
//    Prints the "Hello, World!" message if no operands are
//    present.  Errors out with a Usage message otherwise.
//

#include <libgen.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char **argv) {
   printf ("Hello, World!\n");
   return EXIT_SUCCESS;
}

```

Figure 1. `code/hello.c`

```
# $Id: Makefile,v 1.1 2015-01-13 15:48:01-08 - - $

SOURCES   = hello.c
OBJECTS   = ${SOURCES:.c=.o}
ALLSRCS   = ${SOURCES} Makefile
EXECBIN   = hello

GCC       = gcc -g -O0 -Wall -Wextra -std=gnu11

all: ${EXECBIN}

${EXECBIN}: ${OBJECTS}
        ${GCC} ${OBJECTS} -o ${EXECBIN}

%.o : %.c
        - checksource $<
        cid + $<
        ${GCC} -c $< -o $@

test: ${EXECBIN}

```

Figure 2. **`code/Makefile`**