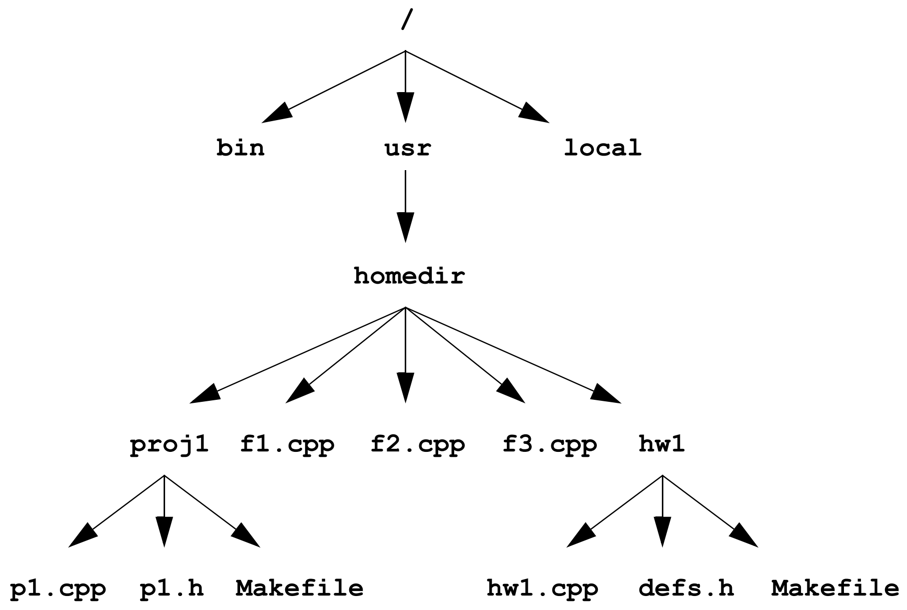
Unix Commands and Utilities

The Unix file system is a tree structure. The root directory for the entire file system is denoted as "/" (a single forward slash). Below the root directory are system sub-directories such as **local** (files specific to the local machine), **bin** (binary executable files), and **misc** (odds and ends). In addition, there are usually directories called **user** or **usr** (**usr1, usr2, usr3** . . .) that contain user accounts. Paths to files and directories are specified using either absolute notation or relative notation. For example, assume your homedir is positioned as shown in the picture below.



The absolute path to the file **p1.cpp** is **/usr/homedir/proj1/p1.cpp**. Assume the current working directory is homedir, a relative path to **p1.cpp** is **./proj1/p1.cpp** or simply **proj1/p1.cpp**. To understand the difference, just remember an absolute path starts at the top of the directory structure (/) and its meaning does not depend on the current working directory. The meaning of a relative path depends on the current working directory. There are three useful shorthand symbols used to specify paths:

. -- is the current directory
.. -- is the parent directory of the current directory
~ -- is the path from the root directory (/) to your home directory

In addition, collections of files and/or directories can be referred to using wildcard notation.

* -- matches anything (*. cpp is the set of all files and directories ending in . cpp)
? -- matches any single character (p?.cpp is the set of all files and directories
of the form: p1.cpp, p2.cpp, pX.cpp etc.)

See the Quick Guide To Unix Commands PDF for a list of a few useful Unix commands

make Utility

The make utility is a software program that helps you compile and link complex programs. The basic idea is that the make utility read commands from a file (typically named makefile, Makefile, or MAKEFILE) that represent instructions about how to compile and link a collection of source files to form a single executable program. There are two terms commonly used to describe makefiles: targets and dependencies (some time referred to as prerequisites). A target is something we are trying to build and dependencies are things that are required to build the target. For example, if we are trying to make an executable file called DOIT from two C++ source files called f1.cpp and f2.cpp, then DOIT is a target that depends on the prerequisites (dependencies) f1.cpp and f2.cpp. The syntax used to specify a target and dependencies is quite simple:

```
target:  dependencies
         command
```

The target is a name (usually a file name) and the dependencies is a list of files that are used to build the target. The target is up-to-date, if the time-stamp on the target is more recent than any of its prerequisite files. When the target is out-of-date, the command on the second line is executed to update the target. (*There is one important format restriction for makefile, commands must begin with a TAB character, otherwise spacing and tabs can be used to seperate items).* Consider the following example.

```
DOIT:    f1.cpp f2.cpp
         g++ -o DOIT f1.cpp f2.cpp
```

If we execute **make** using a makefile containing these two lines, the make utility will automatically check the time-stamp on the files DOIT, f1.cpp, and f2.cpp. If either f1.cpp or f2.cpp are newer (probably because we modified them to make corrections) than DOIT, the **make** utility will execute the command: **g++ -o DOIT f1.cpp f2.cpp** to rebuild the target file DOIT. This will modify the time-stamp on DOIT so if we tried to run **make** again, it will display a message such as "DOIT is up to date".

A makefile actually represents a directed graph structure (see the picture in HW1) that captures the dependency relationships among several files. For example, we can rewrite our DOIT example as follows:

```
DOIT:    f1.o f2.o
         g++ -o DOIT f1.o f2.o

f1.o:    f1.cpp
         g++ -c f1.cpp

f2.o:    f2.cpp
         g++ -c f2.cpp
```

In this modified form, if we execute **make**, the utility will check the time-stamp on f1.o and f2.o. If either of these files is out of date, **make** will locate the command(s) needed to rebuild the appropriate .o file(s). The difference between the first form of the makefile and the second is that if we edit f1.cpp and execute **make**, the first form will recompile both f1.cpp and f2.cpp while the second form will recompile only f1.cpp. This may not seem significant with only two files, but if there were hundreds of files, you would notice the difference.

There are a number of shorthand techniques that can be used to simplify makefiles. First, you can define macros. For example,

```
objs= f1.o f2.o

DOIT:    $(objs)
         g++ -o DOIT $(objs)

f1.o:    f1.cpp
```

```
         g++ -c f1.cpp

f2.o:    f2.cpp
         g++ -c f2.cpp
```

In this example, we are associating the macro name “objs” with the string “f1.o f2.o”. We can insert the macro throughout the makefile by placing a $(. . .) or ${. . .} around the name (e.g. $(objs) or ${objs}). When **make** analyzes this makefile, it will replace each occurrence of a macro name with its corresponding string value. Using macros simplifies the makefile and reduces typing errors. It is very common to see the following form:

```
CC       = g++
CFLAGS   =
LDFLAGS  = -g
LIBS     =
OBJS     = f1.o f2.o

DOIT:    $(OBJS)
         $(CC) $(LDFLAGS) -o DOIT $(OBJS) $(LIBS)

f1.o:    f1.cpp
         $(CC) $(CFLAGS) -c f1.cpp

f2.o:    f2.cpp
         $(CC) $(CFLAGS) -c f2.cpp
```

Notice how we can easily change the compiler, flags, or the set of object files by altering one macro at the top of the makefile. The names (LIBS and CFLAGS) that are associated with empty strings have no effect.

A final simplification allowed in makefile is the use of default rules. For example,

```
CC       = g++
CFLAGS   =
LDFLAGS  = -g
LIBS     =
OBJS     = f1.o f2.o

.SUFFIXES .cpp .o

.cpp .o:
         $(CC) $(CFLAGS) -c $*.cpp

DOIT:    $(OBJS)
         $(CC) $(LDFLAGS) -o DOIT $(OBJS) $(LIBS)

f1.o:    f1.cpp

f2.o:    f2.cpp
```

In this example, the line .SUFFIXES indicates that we will define a default relationship between .cpp files and .o files. After we define the suffixes, we can insert a target of the form .cpp .o with no dependencies. The command for this type of target is $(CC) $(CFLAGS) -c $*.cpp. The $* is a special predefined macro that is replaced by the name of the specific dependent file that triggered the application of the default rule. Notice, we have removed the commands for building f1.o and f2.o. This means the default rule should be used. For example, if f1.cpp is out of date relative to f1.o, then the default rule is invoked using f1.o as the target and f1.cpp as the dependent file. This produces the command: **g++ -c f1.cpp** that is executed to build the target: f1.o. There are a number of special symbols that can be used in makefiles besides $* including $< (a list of dependent files), $? (a list of out-of-date dependent files), and $@ (the current target). For more details about the make utility, type **man make**.

There are a couple of special items to remember:

1. You can add a comment to a makefile by beginning the line with a # character.

2. When you type **make**, the make utility searches the current directory for a file with one of the default names (e. g Makefile). You can override this behavior by entering:

```
make -f yourMakefileName
```

3. The make utility's default behavior is to build the first target in the file. If there are targets that are not prerequisites of the first target or one of its dependent files, they will not be built. You can control exactly what target is built using the command:

```
make target_name
```

   This causes **make** to begin the build operation at a specific target. You can use this feature to include useful commands in the makefile and execute them on demand.   For example, by adding:

```
clean:
         rm -rf *.o DOIT
```

   to the end of our example makefile, we can execute:

```
make clean
```

   which will remove all .o files and the executable file (DOIT) from the current directory. If you execute **make** again without arguments, it will rebuild the default target (i.e. the entire program).