


Johns Hopkins
Engineering for Professionals

**System Development in the UNIX
Environment**

Make and Makefiles


Timothy Barrett



Objectives

- We will explain the need for build tools
- The Make tool will be described and specific techniques will be demonstrated to create and use Makefiles.
- An example project hierarchy will be defined that will be used in future homework assignments for the course.


ep.jhu.edu



Building Complex Systems

- In the early days of software development, most systems consisted of a single monolithic program
- The cycle of 'code-build-test-fix' was trivial to manage
- As hardware evolved, software became more and more complex
 - involved more software 'modules'
 - eventually involved more cooperating programs

ep.jhu.edu



Building Complex Systems

- When dealing with a program built from many software modules, insuring consistency in the build is essential
 - linking old objects in with new objects can produce 'bugs' that are extremely difficult to debug
- Furthermore, when dealing with software systems consisting of many cooperating programs, insuring consistency between the programs is also essential
 - again, is very difficult to debug

ep.jhu.edu



Building Complex Systems

- One solution is to re-build the entire system from its source code each time the system software is modified.
 - Very expensive, can take a lot of time
- A better solution is to use a tool that can rebuild only the parts of the system software that were modified or depend on the modified software
 - the tool **must** be trustworthy!

ep.jhu.edu



Make

- Make is a UNIX utility that has been in use since the early days of UNIX.
- Make is dependent on file modification times.
- This utility is most often used in building a program based on its source code (for example, updating a '.o' file if the corresponding '.c' file is newer).
- Make uses a description file (Makefile) to control how make goes about building the output

ep.jhu.edu



Makefiles

- A *Makefile* is a description file that **make** uses to build targets. This description file includes:
 - A list of *targets* and *dependencies* with associated commands to build the targets
 - Macros
 - Suffix Rules
- By default, the description file is named 'Makefile' or 'makefile'. The default can be overridden by the '-f' flag to the **make** command.

ep.jhu.edu



Makefiles

- The makefile description file syntax:
 - Dependency lines: These line contain a colon. To the left of the colon are one or more targets; to the right of the colon are the component files on which the targets depend.

```
target: target.o util.o
```

target depends on target.o and util.o

ep.jhu.edu



Makefiles

- The makefile description file syntax:
 - Command lines: These lines describe how to build the targets. These lines must begin with a tab character.

```
target: target.o util.o
^      cc target.o util.o -o target
^      @echo target is built
tab
```

ep.jhu.edu



-o = output?

example: class_notes/unix_system/make/.../Make1

to check: make -n
to run: make

Makefiles

- The makefile description file syntax:
 - Macro definition lines: These lines define a macro in the form “MACRODEFN=macrovalue”
OBJS = target.o utils.o
 - A macro's value is accessed by using a '\$':
target: \$(OBJS)
 - '#' is the comment delimiter; text from a '#' to the end of the line is ignored.
 - Blank lines are ignored.

ep.jhu.edu



A sample Makefiles

```
program: main.o sortit.o mergeit.o
    cc -o program main.o sortit.o mergeit.o

main.o: main.c
    cc -c main.c
sortit.o: sortit.c
    cc -c sortit.c
mergeit.o: mergeit.c
    cc -c mergeit.c
main.o: header.h
```

ep.jhu.edu



A Sample Makefile with Macros

```
OBJS=main.o sortit.o mergeit.o
program: $(OBJS)
    cc -o program $(OBJS)

main.o: main.c
    cc -c main.c
sortit.o: sortit.c
    cc -c sortit.c
mergeit.o: mergeit.c
    cc -c mergeit.c
main.o: header.h
```

ep.jhu.edu



More on Makefile Macros

- A macro definition is a line containing an equals sign (=). That line cannot be preceded by a space or a tab.
- Macro definitions are fixed before command execution begins.
 - Command execution does not expand macros until the command is executed.
 - Don't redefine macros you use in target dependency lines.
- Undefined macros are assigned the NULL string.
- It is allowed and often useful to use macros in macro definitions.

ep.jhu.edu



Useful Makefile Notes

- Values in the Makefile may be obtained from the environment. (ex. `setenv CC gcc`)
- Make `-e` will force a Makefile to use the environment variables over those defined in the file.
- When creating Makefiles using the `-n` flag is helpful. This will cause the make to print what will be executed, however will not actually compile the code. (`make -n`)
- To look at the default rules use (`make -print-data-base`)

ep.jhu.edu



More on Makefile Macros

- The value of the macro ABC is referenced by `$(ABC)` or `${ABC}`. The delimiters are optional on single character macros (however, single character macros are NOT recommended in most cases).
- Macros can be defined (in priority order):
 - on the make command line
 - in the make description file (Makefile)
 - in the environment
 - by default

ep.jhu.edu



Internal Makefile Macros

- `$?` - list of components that are out of date with respect to the current target

```
print: doc1 doc2 doc3
    nroff $?; touch print
```

- `$$` - Current target name

```
part1 part2: para1 para2 para3
    cat $$ > $$
```

ep.jhu.edu



Suffix Rules

- There are a default set of rules to determine how to build a file with a particular suffix from a file with some other suffix. This is called a '*suffix rule*'.
- Suffix rules are described as a target line with no dependency; the target is a combination of the source suffix followed by the target suffix:

```
.{src}.{target}: would describe how to build files
with the {target} suffix from files that have a {src}
suffix.
```

ep.jhu.edu



Special Macros in Suffix Rules

- There are some macros that are only valid within suffix rules:
 - `$<` - the list of components that are out of date with respect to the current target (similar to `$?`)
 - `$*` - the current target name without the suffix.

ep.jhu.edu



Suffix Rules

- One suffix rule for example, describes how to build .o files from .c files:

```
.c.o:  
cc -c $<
```

- The suffixes provided can be more or less than one character:

```
.cc.o: - how to build .o files from .cc files.  
.c: - how to build files with no suffix from .c files.
```

ep.jhu.edu



Suffix Rules

- Many suffixes are recognized by make by default. Other suffixes can be added by defining the special target '.SUFFIXES'

```
.SUFFIXES: .out  
.c.out:  
cc $< -o $*.out
```

- Order of the suffix in the .SUFFIXES target is important; it will determine which suffix rule applies if there is an ambiguity.

ep.jhu.edu



A Sample Makefile with Macros and Suffix Rules

```
OBJS=main.o sortit.o mergeit.o printit.o  
program: $(OBJS)  
cc -o program $(OBJS)
```

```
.c.o: to build .o from .c, use this command  
cc -c $<
```

```
main.o: header.h
```

ep.jhu.edu



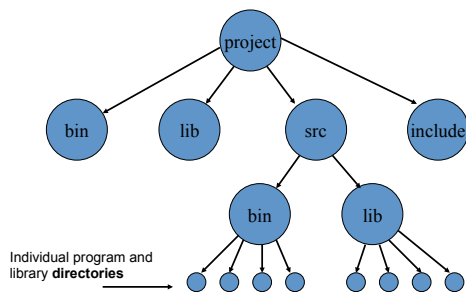
Using Make in a Software Project

- Make (or something equivalent) is an essential tool in any software system that is made up of several programs that use project libraries and include files.
- A software system should be organized into programs and libraries in a hierarchical directory structure.

ep.jhu.edu



System Software Organization



ep.jhu.edu



System Software Organization

- project - the root of the system. (Will be named according to the project.)
- project/bin - the directory containing installed system executables
- project/lib - the directory containing installed system libraries (*.a files)
- project/include - the directory containing header files shared between libraries and executables.

ep.jhu.edu



Software System Organization

- project/src - directory containing all system source code
- project/src/lib - directory containing system source code for the libraries.
- project/src/bin - directory containing system source code for the executables.
- project/src/lib/xyz - directory containing source code for library xyz (creates libxyz.a)
- project/src/bin/abc - directory containing source code for executable abc (creates abc)

ep.jhu.edu



Using Make in a Software Project

- There should be Makefiles in each directory of the project hierarchy.
- If all Makefiles are structured properly (i.e. they include the proper targets which perform the appropriate functions), one function can be performed throughout the directory tree by issuing the appropriate make command in the project root directory.

ep.jhu.edu



Using Make in a Software Project

- Several targets that would be useful to be included in all project Makefiles include:
 - **it** - This target will depend on the eventual program or library. This allows all programs and libraries to be rebuilt by issuing a 'make it' at the src or project level.
 - **install** - Makes 'it' and then installs the program or library in the project bin or lib.
 - **clean** - removes all object files from the directory and low level executables.
 - **depend** - provides automatic generation of Makefile dependencies

ep.jhu.edu



recompiles everything below,
but leaves .a at bottom level (i.e.
libtemp_conv.a is in temp_conv
directory)

Macro String Substitution

- Some versions of make have the ability to create a new macro based on string substitution on an existing macro.
- String substitution can only take place at the end of the macro, or immediately before white space.

```
SRCS=file1.c file2.c file3.c
```

```
OBJS=$(SRCS:.c=.o) use OBJS to get all .o files
```

ep.jhu.edu



Make “depend”

- If a program contains several include files, and those include files contain include files, and so forth, describing those dependencies can be a difficult task to perform correctly.
- Therefore, a **depend** target that automatically generates makefile dependencies assures that your programs will be up to date.
- On many UNIX systems, switches have been added to C or C++ compilers to generate Makefile dependencies for programs.

ep.jhu.edu



Make “depend”

- Some UNIX systems provide a ‘makedep’ command which will generate makefile dependencies.
- Some UNIX systems have added a ‘.KEEP_STATE’ target to their make utility. If this target is present, make keeps track of module dependencies, both explicit and hidden. .KEEP_STATE is not standard, however, and is not supported in all versions of UNIX.

ep.jhu.edu



Useful Switches on the Compiler

- The -xM flag switch generates Makefile dependencies for the Solaris ANSI C and C++ compilers
- The -I<path_name> adds the directory <path_name> to the list of directories to search for include files for compiling.
- The -L<path_name> adds the directory <path_name> to the list of directories to search for library files for linking.

ep.jhu.edu