cs3157: sh and makefile lecture (mon-28-mar-2005)

- today:
  - sh scripts
  - makefiles
  - software engineering basics

---

sh (1)

- `sh` is the “Bourne shell”, the first scripting language
- it is a program that interprets your command lines and runs other programs
- it can invoke Unix commands and also has its own set of commands
- example:

```
while ( 1 ) {
  print prompt and wait for user to enter input;
  read input from terminal;
  parse into words;
  substitute variables;
  execute commands (execv or builtin);
}
```

---

sh (2)

- shell commands can be read:
  - from a terminal ⇒ *interactive*
  - from a file ⇒ *shell script*
- search path
  - the place where the shell looks for the commands it runs
  - should include standard directories:
    - `/bin`
    - `/usr/bin`
  - it should also include your current working directory ()

---

sh (3)

- are you runnning the Bourne shell?
  - type:
    `unix-prompt# echo $SHELL`
  - if the answer is `/bin/sh`, then you are
  - if the answer is `/bin/bash`, then that’s close enough
  - otherwise, you can start the Bourne shell by typing `sh` at the UNIX prompt

  - enter `Ctrl-D` or `exit` to exit the Bourne shell and go back to whatever shell you were running before...

sh (4)

- capable of both synchronous and asynchronous execution
  - synchronous: wait for completion
  - asychronous: in parallel with shell (runs in the background)
- allows control of `stdin, stdout, stderr`
- enables environment setting for processes (using inheritance between processes)
- sets default directory

---

sh (5)

- creating your own shell scripts
- naming:
  - DON’T ever name your script (or any executable file) “test”
  - since that’s a `sh` command
- executing
  - the notation `#!` inside your file tells UNIX which shell should execute the commands in your file
- example — create a file called “myscript.sh”

```
#!/bin/sh
echo hello world
```

- make the script executable: `unix-prompt# chmod +x myscript.sh`
- execute the script: `unix-prompt# ./myscript.sh` or just `unix-prompt# myscript.sh`

(note that `unix-prompt#` *means the unix prompt, like* `unix$` *or* `bash#`*)*

---

sh (6) — quoting

- quote (`’`)
  `’something’`: preserve everything literally and don’t evaluate anything that is inside the quotes
- double quote (`"`)
  `"something"`: preserve most things literally, but also allow `$` variable expansion (but not `’` evaluation)
- backquote (`‘`)
  `‘something‘`: try to execute `something` as a command

---

sh (7) — quoting example

- filename=t.sh

```
#!/bin/sh
hello="hi"
echo 0=$hello
echo 1=’$hello’
echo 2="$hello"
echo 3=‘$hello‘
echo 4="‘$hello‘"
echo 5="’$hello’"
```

- filename=hi

```
#!/bin/sh
echo "how did you get in here?"
```

- output=

```
unix$ t.sh
0=hi
1=$hello
2=hi
3=how did you get in here?
4=how did you get in here?
5=’hi’
```

sh (8) — comments

- single line comments only (no multi-line comments)
- line begins with # character

---

sh (9) — simple commands

- sequence of words
- first word defines command
- can be combined with `&&`, `||`, `;`
  - to execute commands sequentially:
    `cmd1; cmd2;`
  - to execute a command in the background :
    `cmd1&`
  - to execute two commands asynchronously:
    `cmd1&`
    `cmd2&`
  - to execute `cmd2` if `cmd1` has zero exit status:
    `cmd1 && cmd2`
  - to execute `cmd2` only if `cmd1` has non-zero exit status:
    `cmd1 || cmd2`
- set exit status using `exit` command (e.g., `exit 0` or `exit 1`)

---

sh (10) — pipes

- sequence of commands
- connected with `|`
- each command reads previous command's output and takes it as input
- example:

```
unix-prompt# echo "hello world" | wc -w
2
```

---

sh (11) — shell variables

- variables are placeholders for values
- shell does variable substitution
- `$var` or `${var}` is the value of the variable
- assignment:
  - `var=value` (with no spaces before or after!)
  - `let "var = value"`
  - `export var=value`
- BUT values go away when shell is done executing
- uninitialized variables have no value
- variables are untyped, interpreted based on context
- standard shell variables:
  - `${N}` = shell Nth parameter
  - `$$` = process ID
  - `$?` = exit status

sh (12) — shell variables example

- filename=u.sh

```
#!/bin/sh
echo 0=$0
echo 1=$1
echo 2=$2
echo 3=$$
echo 4=$?
```

- output

```
unix$ u.sh
0=.//u.sh
1=
2=
3=21093
4=0

unix$ u.sh abc 23
0=.//u.sh
1=abc
2=23
3=21094
4=0
```

---

sh (13) — environment variables

- shell variables are generally not visible to programs
- environment variables are a list of name/value pairs passed to sub-processes
- all environment variables are also shell variables,
  but not vice versa
- show with `env` or `echo $var`
- standard environment variables include:
  - HOME = home directory
  - PATH = list of directories to search
  - TERM = type of terminal (vt100, ...)
  - TZ = timezone (e.g., US/Eastern)
- example:

```
unix-prompt# echo $TERM
vt100
```

---

sh (14) — looping constructs

- similar to C/Java constructs, but with commands
- `until test-commands; do consequent-commands; done`
- `while test-commands; do consequent-commands; done`
- `for name [in words ...]; do commands; done`
- also on separate lines
- `break` and `continue` control loop

---

sh (15) — loop examples

- while

```
i=0
while [ $i -lt 10 ]; do
  echo "i=$i"
  ((i=$i+1))   # same as let "i=$i+1"
done
```

- for

```
for counter in ‘ls *.c‘; do
  echo $counter
done
```

sh (16) — if

- syntax

```
if test-commands; then
        consequent-commands;
        [elif more-test-commands; then
                more-consequents;]
        [else alternate-consequents;]
fi
```

- colon (`:`) is a null command
- example

```
#!/bin/sh
if expr $TERM = "xterm"; then
  echo "hello xterm";
else
  echo "something else";
fi
```

---

sh (17) — case

- example:

```
case test-var in
value1) consequent-commands;;
value2) consequent-commands;;
*) default-commands;
esac
```

- pattern matching:
  - `?)` matches a string with exactly one character
  - `?*)` matches a string with one or more characters
  - `[yY]|[yY][eE][sS])` matches `y, Y, yes, YES, yES`...
  - `/*/*[0-9])` matches filename with wildcards like `/xxx/yyy/zzz3`
  - notice two semi-colons at the end of each clause
  - stops after first match with a value
  - you don’t need double quotes to match string values!

---

sh (18) — case example

```
#!/bin/sh
case "$TERM" in
xterm) echo "hello xterm";;
vt100) echo "hello vt100";;
*) echo "something else";;
esac
```

---

sh (19) — expansion

- biggest difference from traditional programming languages
- shell substitutes and executes
- order:
  - brace expansion
  - tilde expansion
  - parameter and variable expansion
  - command substitution
  - arithmetic expansion
  - word splitting
  - filename expansion

sh (20) — brace expansion

- expand comma-separated list of strings into separate words:

```
unix-prompt# echo a{d,c,b}e
ade ace abe
```

- useful for generating list of filenames:

```
unix-prompt# mkdir hw{1,2,3}
unix-prompt# ls
hw1  hw2  hw3
```

---

sh (21) — tilde expansion

- `˜` expands to `$HOME`
- examples:
  `˜cs3157` ⇒ `/home/cs3157`
  `˜/html` ⇒ `/home/sklar/html`
- but doesn’t always work inside a shell script

---

sh (22) — command substitution

- replace `$(command)` or `‘command‘` by stdout of executing command
- can be used to execute content of variables:

```
unix$ x=ls
unix$ $x
myfile.c
a.out
unix$ echo $x
ls
unix$ echo ‘ls‘
myfile.c
a.out
unix$ echo ‘x‘
sh: x: command not found
unix$ echo ‘$x‘
myfile.c
a.out
unix$ echo $(ls)
myfile.c
a.out
unix$ echo $(x)
sh: x: command not found
unix$ echo $($x)
myfile.c
a.out
```

---

sh (23) — filename expansion

- any word containing `*?([` is considered a *pattern*
- `*` matches any string
- `?` matches any single character
- `[...]` matches any of the enclosed characters

```
unix$ ls
myfile.c
a.out
a.b
unix$ ls a*
a.out
a.b
unix$ ls a?
ls: No match.
unix$ ls a.*
a.out
a.b
unix$ ls a.?
a.b
unix$ ls a.???
a.out
unix$ ls [am].b
a.b
```

sh (24) — redirections

- `stdin`, `stdout` and `stderr` may be redirected
- `<` redirects `stdin` (0) to come from a file
- `>` redirects `stdout` (1) to go to file
- `>>` appends `stdout` to the end of a file
- `&>` redirects `stderr` (2)
- `>&` redirects `stdout` and `stderr`, e.g.: `2>&1` sends `stderr` to the same place that `stdout` is going
- `<<` gets input from a *here document*, i.e., the input is what you type, rather than reading from a file

built-in sh commands (1)

- `alias, unalias` — create or remove a pseudonym or shorthand for a command or series of commands
- `jobs, fg, bg, stop, notify` — control process execution
- `command` — execute a simple command
- `cd, chdir, pushd, popd, dirs` — change working directory
- `echo` — display a line of text
- `history, fc` — process command history list
- `set, unset, setenv, unsetenv, export` — shell built-in functions to determine the characteristics for environmental variables of the current shell and its descendents
- `getopts` — parse utility options
- `hash, rehash, unhash, hashstat` — evaluate the internal hash table of the contents of directories
- `kill` — send a signal to a process

built-in sh commands (2)

- `pwd` — print name of current/working directory
- `shift` — shell built-in function to traverse either a shell's argument list or a list of field-separated words
- `readonly` — shell built-in function to protect the value of the given variable from reassignment
- `source` — execute a file as a shell script
- `suspend` — shell built-in function to halt the current shell
- `test` — check file types and compare values
- `times` — shell built-in function to report time usages of the current shell
- `trap, onintr` — shell built-in functions to respond to (hardware) signals
- `type` — write a description of command type
- `typeset, whence` — shell built-in functions to set/get attributes and values for shell variables and functions

built-in sh commands (3)

- `limit, ulimit, unlimit` — set or get limitations on the system resources available to the current shell and its descendents
- `umask` — get or set the file mode creation mask

what is make?

- utility typically used for building software packages that are comprised of many source files
- determines automatically which pieces need to be rebuilt
- uses an input file (usually called `makefile` or `Makefile`) which specifies *rules* and *dependencies* for building each piece
- you can use any name for the makefile and specify it on the command line:

```
unix-prompt# make
unix-prompt# make -f myfile.mk
```

- first way (above) uses default (`makefile` or `Makefile`) as input file
- second way uses `myfile.mk` as input file

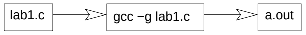
---

make tutorial (1)

- Let's begin by considering the simplest case of compiling a C program.
- Suppose that you have a C program called `lab1.c`.
- If you were going to compile this on the command line using the `gcc` compiler and send the output to the default file `a.out`, you'd execute the following command:

```
unix-prompt# gcc lab1.c
```

- This is illustrated in the figure below:



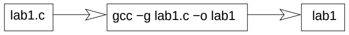
---

make tutorial (2)

- Now suppose you don't want to use the default output file name, but instead you want to name the output executable `lab1`.
- Then you would execute the following command:

```
unix-prompt# gcc lab1.c -o lab1
```

- This is illustrated in the figure below:



---

make tutorial (3)

- Next, suppose that you have a more complicated case — you have two C source files (`hw1.c` and `inv.c`), and you want to compile them and link them together to create one executable program called `hw1`.
- This is a multi-step process. First, you need to compile each source file into object code:

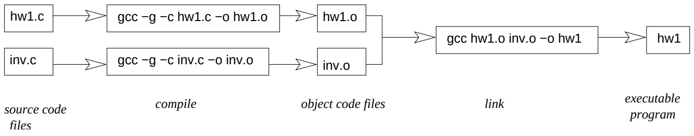
```
unix-prompt# gcc -c hw1.c -o hw1.o
unix-prompt# gcc -c inv.c -o inv.o
```

  The `-c` switch on the gcc command tells gcc to *compile only* and not link. So, after these two commands are executed, you have two object code files (`hw1.o` and `inv.o`). These files are not executable — they must be linked.
- We'll assume that `hw1.c` refers to components in `inv.c`, indicating that the files must be linked together. For example, `inv.c` contains the definition of a function called `printInventory()`, and `hw1.c` contains a call to that function. Thus `hw1.c` needs `inv.c` to resolve its "external references". Similarly, `hw1.c` contains a `main()` function, but `inv.c` doesn't; so the files must be linked together — or at least, `inv.c` needs to be linked to some file that contains a `main()`.

- The link step is as follows:

```
unix-prompt# gcc hw1.o inv.o -o hw1
```

- This is illustrated in the figure below:



---

make tutorial (4)

- The advantage of using a `makefile` is that it will keep track of which files need to be compiled when building an executable program that takes more than one source file.
- It is also easier to compile using make, because at the unix command line, all you have to type is:

```
unix-prompt# make
```

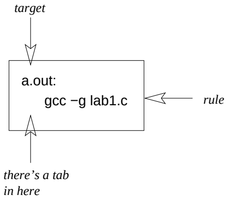
- Note that sometimes you will also type a target as an argument to make, such as:

```
unix-prompt# make step1
```

---

make tutorial (5)

- Just as above we started simple and added complexity, let's start with a very simple `makefile` and build from it. The figure below shows a simple example. The simple makefile has two components: a *target* and a *rule*.
- When you type `make` at the unix prompt, the make utility reads the `makefile` and executes the *rule* associated with the `a.out` *target*, namely it does exactly the same thing that is done on make tutorial page (1).

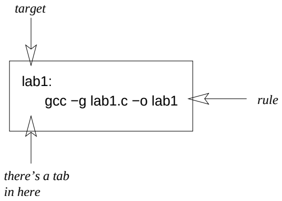


```
a.out:
	gcc −g lab1.c
```

---

make tutorial (6)

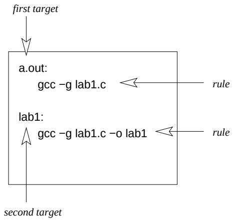
- As above in going from "make tutorial pages (1) to (2), we add an output file name to the simple command, shown in the makefile in the figure below. The execution is the same.
- When you type `make` at the unix prompt, the make utility reads the `makefile` and executes the *rule* associated with the `lab1` *target*.



```
lab1:
	gcc −g lab1.c −o lab1
```

make tutorial (7)

- In both of the simple `makefile` examples above, there is only one *target*. If there were more than one target, then typing `make` with no arguments would cause the `make` utility to execute the rule associated with the first target that appears in the file.
- For example, in the figure below:
  - Typing `make`, would build target `a.out`.
  - Typing `make a.out`, would also build target `a.out`.
  - Typing `make lab1`, would build target `lab1`.

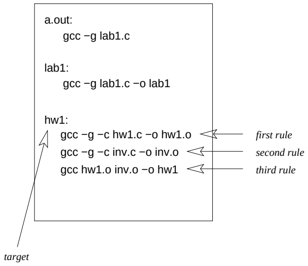


```
a.out:
    gcc −g lab1.c

lab1:
    gcc −g lab1.c −o lab1
```

---

make tutorial (8)

- Okay, now let’s move on to building a program that has two source files, as in make tutorial page (3).
- We add a target, `hw1`, to our makefile, as shown in the figure below. This new target has three rules, which get executed consecutively, just as if they were typed on the unix command line one after the other. To execute this target, type `make hw1` on the unix command line.

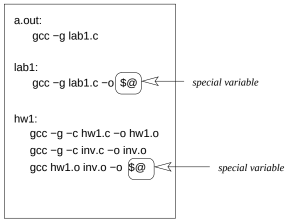


```
a.out:
    gcc −g lab1.c

lab1:
    gcc −g lab1.c −o lab1

hw1:
    gcc −g −c hw1.c −o hw1.o
    gcc −g −c inv.c −o inv.o
    gcc hw1.o inv.o −o hw1
```

---

make tutorial (9)

- Up to this point, we haven’t really taken advantage of the power of the `make` utility, except to save ourselves some typing. Next, let’s look at some of the features of `make`. We’ll start by introducing some special variables.
- The variable `$@` is used inside a rule and it stands for the name of the target that the rule is associated with. For example, we could replace the `-o lab1` portion of the rule for the `lab1` target with `-o $@`. The meaning would be exactly the same. The figure below is the same makefile on make tutorial page (8), but uses the `$@` special variable.

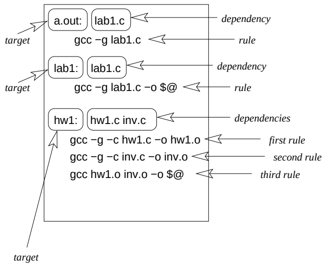


```
a.out:
    gcc −g lab1.c

lab1:
    gcc −g lab1.c −o $@

hw1:
    gcc −g −c hw1.c −o hw1.o
    gcc −g −c inv.c −o inv.o
    gcc hw1.o inv.o −o $@
```

- Note that we don’t use this special variable with the first target’s rule, because the name of the first target (`a.out`) is not speciifed in that target’s rule.
- Also note that we don’t use the special variable in the first two rules belonging to the `hw1` target, again because the name of the target (`hw1`) is not specified in either of these rules. Using the special variable does not change the way the makefile is executed.
- You would still, for example, type `make lab1` to build the second target.

make tutorial (10)

- Now let's talk about *dependencies*, which is one of the really nifty things about `make`. A dependency is something that tells `make` whether or not it needs to execute the rules associated with a target. Dependencies are listed on the same line as a target, after the colon (`:`) which follows the name of the target. Multiple dependencies are separated by spaces.
- In the figure below, our `makefile` includes dependencies for all three targets.



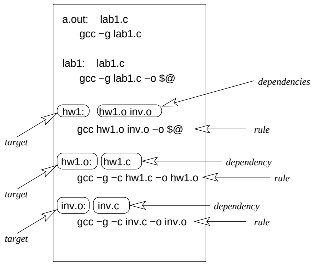
```
a.out:  lab1.c
     gcc −g lab1.c

lab1:  lab1.c
     gcc −g lab1.c −o $@

hw1:  hw1.c inv.c
     gcc −g −c hw1.c −o hw1.o
     gcc −g −c inv.c −o inv.o
     gcc hw1.o inv.o −o $@
```

- For the first target (`a.out`), the dependency listed is `lab1.c`.
- When make executes to build this target, it compares the "last modified" date of the target file (if it exists) with the last modified date of its dependency. If the dependency is *newer* than its target, then the target's rule is executed. If a file bearing the same name as the target doesn't exist, then the target's rule is executed as well. The same goes for the second target. If the file `lab1` exists and it is older than its dependency, `lab1.c`, or if `lab1` doesn't exist, then the target's rule is executed.
- For the third target (`hw1`), there are two dependencies: `hw1.c` and `inv.c`. If either one of these files is newer than `hw1`, or if `hw1` doesn't exist, then the target's three rules are all executed.
- Adding dependencies doesn't change the way the `makefile` is executed. You would still type `make a.out` or `make lab1` or `make hw1` to build each of the three targets.

---

make tutorial (11)

- Now, let's examine this third target a little more closely. To be more precise, the target itself, `hw1`, actually depends directly on `hw1.o` and `inv.o`, not the C source files.
- In addition, if you edit `hw1.c` but not `inv.c` since the last time you built the target, then you really only need to recompile `hw1.c`, not both `hw1.c` and `inv.c`.
- So let's split this up into its three constituent rules, as illustrated in the figure below.

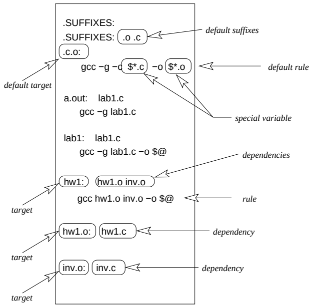


```
a.out:  lab1.c
     gcc −g lab1.c

lab1:  lab1.c
     gcc −g lab1.c −o $@

hw1:  hw1.o inv.o
     gcc hw1.o inv.o −o $@

hw1.o:  hw1.c
     gcc −g −c hw1.c −o hw1.o

inv.o:  inv.c
     gcc −g −c inv.c −o inv.o
```

- Doing this adds two targets to the makefile. This means that in addition to being able to type `make a.out` or `make lab1` or `make hw1` to build each of the three original targets, you could also build either of the two "intermediate" targets by typing `make hw1.o` or `make inv.o`.
- These are referred to as "intermediate" because building these two targets only results in updated object code, not executable programs.

make tutorial (12)

- You probably noticed in the figure on the previous page that the rules for the last two targets are very similar. Indeed, they are identical except for the file names. This is where *default rules* come in handy.
- In the figure below, the *rule* portions of the last two targets are removed and replaced by the single *default rule* at the top of the file.



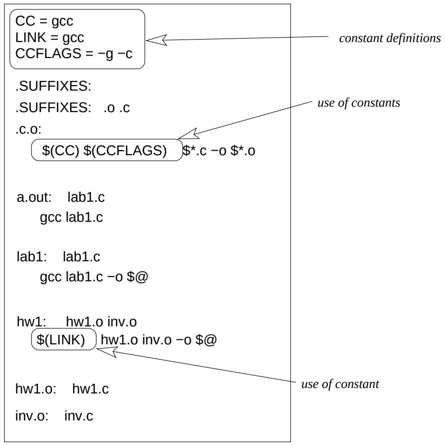
```
.SUFFIXES:
.SUFFIXES: .o .c
.c.o:
    gcc −g −c $*.c −o $*.o

a.out:   lab1.c
    gcc −g lab1.c

lab1:   lab1.c
    gcc −g lab1.c −o $@

hw1:   hw1.o inv.o
    gcc hw1.o inv.o −o $@

hw1.o:  hw1.c

inv.o:  inv.c
```

- The default rule has two parts to it:
  1. The *SUFFIXES* define which file extensions have default targets associated with them.
  2. The *default target* is listed with its associated *default rule*. In this example, the *default target* gives a default rule for building any `.o` file out of a `.c` file.
- The default rule uses another special variable: `$*`. This variable stands for the *filename* portion of the dependency that invoked the rule. In other words, if `hw1.c` invoked the rule (because it was newer than its target `hw1.o`), then the special variable `$*` would take on the value `hw1`. Similarly, if `inv.c` invoked the rule (because it was newer than its target `inv.o`), then the special variable `$*` would take on the value `inv`.
- Note that these changes do not affect the way the `makefile` is executed. Default targets cannot be built directly by specifying them on the command line, so we still have 5 targets that can be built with this makefile; and each of these targets is specified in the same way as in the figure on make tutorial page (11).

---

make tutorial (13)

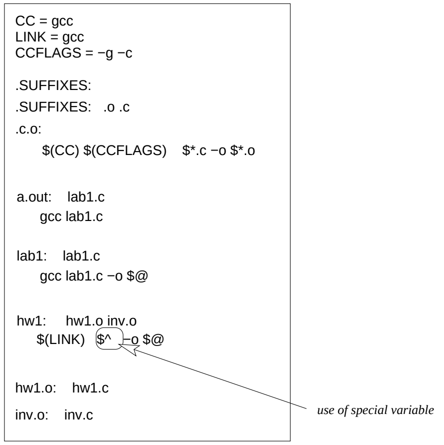
- Another feature of make is the ability to *user-defined constants*.
- The example shown in the figure below illustrates the use of three user-defined constants:
  - `CC` (which stands for the C Compiler)
  - `LINK` (which stands for the Linker)
  - `CCFLAGS` (which contains the flags to be used when the C Compiler is invoked)
- Note that the C Compiler and the Linker are actually the same program (`gcc`), but defining them separately provides the flexibility to use different programs for each if we wanted to do so.



```
CC = gcc
LINK = gcc
CCFLAGS = −g −c

.SUFFIXES:
.SUFFIXES:   .o .c
.c.o:
    $(CC) $(CCFLAGS) $*.c −o $*.o

a.out:    lab1.c
    gcc lab1.c

lab1:    lab1.c
    gcc lab1.c −o $@

hw1:     hw1.o inv.o
    $(LINK) hw1.o inv.o −o $@

hw1.o:    hw1.c
inv.o:    inv.c
```

make tutorial (14)

- The last feature of make that we have used is the special variable `$^`, which is used in a rule to stand for the list of a target's dependencies.
- This illustrated in the figure below.
- When make executes the target where the special variable is indicated, the value of `$^` is replaced with `hw1.o inv.o`, the list of dependencies which belong to that rule's target.



```
CC = gcc
LINK = gcc
CCFLAGS = −g −c

.SUFFIXES:
.SUFFIXES:   .o .c
.c.o:
      $(CC) $(CCFLAGS)    $*.c −o $*.o


a.out:    lab1.c
      gcc lab1.c


lab1:    lab1.c
      gcc lab1.c −o $@


hw1:     hw1.o inv.o
     $(LINK)   $^   −o $@


hw1.o:    hw1.c
inv.o:    inv.c
```

use of special variable

make: defining rules

- the syntax is:

```
<target> : <dependencies>
     <tab><command1>
     <tab><command2>
     ...
     <tab><commandN>
```

- there must be a `<tab>` at the beginning of each command line
- for example:

```
foo.o : foo.c defs.h       # rule for building foo.o
     cc -c -g foo.c
```

make: specifying targets

- you can specify a target on the command line:

```
unix-prompt# make -f myfile.mk install
```

- the default target is the first one in the makefile (i.e., if you don't specify a target on the command line)
- often you have the following targets:
  - `all`
  - `clean`
  - `install`

make: wildcards

- wildcard characters are `*`, `?` and `[...]` are the same as in the Bourne shell
- variables are also like in the Bourne shell (i.e., begin with `$`)
- but be careful because environment variables are imported into make
- there are a number of automatic variables:
  - `$@` = the file name of the rule target
  - `$?` = names of all dependencies that are newer than the target
  - `$ˆ` = names of all dependencies
- you can also use `F` and `D` to get the file and directory (respectively) portions of full paths
- e.g., `$(@D)` and `$(@F)` return the directory and file names of the target

---

make: example

- example:

```
LIB  = $(HOME)/lib
INC  = $(HOME)/include
BIN  = $(HOME)/bin

RCS      = RCS
CC       = gcc
LINK     = gcc
CCFLAGS  = -c -g
```

- defines many variables
- which are refered to like this, e.g.: `$(CC)`
- notice use of `$(HOME)` which is read from the environment

---

make: implicit rules

- *implicit* rules can be used to define a general way of building one type of file from another
- for example

```
.SUFFIXES:
.SUFFIXES: .o .c

.c.o:
     $(CC) $(CCFLAGS) $*.c -o $*.o
```

- note use of variables

---

make: dependencies

- it is good practice to list include files as dependencies
- for example:

```
hw4sklarserver: hw4sklar.o util.o
     $(LINK) $(LDFLAGS) -o $@ $ˆ

hw4sklar.o: hw4sklar.c hw4sklar.h
util.o: util.c util.h
```

- this will use the implicit rule to know how to build a `.o` file from a `.c` file

software engineering: what is it?

- Stephen Schach: “Software engineering is a discipline whose aim is the production of fault-free software, delivered on time and within budget, that satisfies the user’s needs.”
- includes:
  - requirements analysis
  - human factors
  - functional specification
  - software architecture
  - design methods
  - programming for reliability
  - programming for maintainability
  - team programming methods
  - testing methods
  - configuration management

---

software engineering: why?

- in school, you learn the *mechanics* of programming
- you are given the specifications
- you know that it is possible to write the specified program in the time allotted
- but not so in the real world...
  - what if the specifications are not possible?
  - what if the time frame is not realistic?
  - what if you had to write a program that would last for 10 years?
- in the real world:
  - software is usually late, overbudget and broken
  - software usually lasts longer than employees or hardware
- the real world is cruel and software is fundamentally brittle

---

software engineering: who?

- the average manager has no idea how software needs to be implemented
- the average customer says: “build me a system to do X”
- the average layperson thinks software can do anything (or nothing)
- most software ends up being used in very different ways than how it was designed to be used

---

software engineering: time.

- you never have enough time
- software is often underbudgeted
- the marketing department always wants it tomorrow
- even though they don’t know how long it will take to write it and test it
- “Why can’t you add feature X? It seems so simple...”
- “I thought it would take a week...”
- “We’ve got to get it out next week. Hire 5 more programmers...”

software engineering: people.

- you can’t do everything yourself
- e.g., your assignment: “write an operating system”
- where do you start?
- what do you need to write?
- do you know how to write a device driver?
- do you know what a device driver is?
- should you integrate a browser into your operating system?
- how do you know if it’s working?

software engineering: complexity.

- software is complex!
- or it becomes that way
  - feature bloat
  - patching
- e.g., the evolution of Windows NT
  - NT 3.1 had 6,000,000 lines of code
  - NT 3.5 had 9,000,000
  - NT 4.0 had 16,000,000
  - Windows 2000 has 30-60 million
  - Windows XP has at least 45 million...

software engineering: necessity.

- you will need these skills!
- risks of faulty software include
  - loss of money
  - loss of job
  - loss of equipment
  - loss of life

examples: therac-25 (1).

- `http://sunnyday.mit.edu/papers/therac.pdf`
- therac-25 was a linear accelerator released in 1982 for cancer treatment by releasing limited doses of radiation
- it was software-controlled as opposed to hardware-controlled (previous versions of the equipment were hardward-controlled)
- it was controlled by a PDP-11; software controlled safety
- in case of error, software was designed to prevent harmful effects

examples: therac-25 (2).

- BUT
- in case of software error, cryptic codes were displayed to the operator, such as:
  “MALFUNCTION xx”
  where $1 < xx < 64$
- operators became insensitive to these cryptic codes
- they thought it was impossible to overdose a patient
- however, from 1985-1987, six patients received massive overdoses of radiation and several died

examples: therac-25 (3).

- main cause:
- a race condition often happened when operators entered data quickly, then hit the up-arrow key to correct the data and the values were not reset properly
- the manufacturing company never tested quick data entry — their testers weren’t that fast since they didn’t do data entry on a daily basis
- apparently the problem had existed on earlier models, but a hardware interlock mechanism prevented the software race condition from occurring
- in this version, they took out the hardware interlock mechanism because they trusted the software

examples: ariane 501 (1).

- next-generation launch vehicle, after ariane 4
- presigious project for ESA
- maiden flight: june 4, 1996
- inertial reference system (IRS), written in ada
  - computed position, velocity, acceleration
  - dual redundancy
  - calibrated on launch pad
  - relibration routine runs after launch (active but not used)
- one step in recalibration converted floating point value of horizontal velocity to integer
- ada automatically throws out of bounds exception if data conversion is out of bounds
- if exception isn’t handled... IRS returns diagnostic data instead of position, velocity, acceleration

examples: ariane 501 (2).

- perfect launch
- ariane 501 flies much faster than ariane 4
- horizontal velocity component goes out of bounds
- IRS in both main and redundant systems go into diagnostic mode
- control system receives diagnotic data but interprets it as wierd position data
- attempts to correct it...
- ka-boom!
- failure at altitiude of 2.5 miles
- 25 tons of hydrogen, 130 tons of liquid oxygen, 500 tons of solid propellant

examples: ariane 501 (3).

- expensive failure:
  - ten years
  - $7 billion
- horizontal velocity conversion was deliberately left unchecked
- who is to blame?
- “mistakes were made”
- software had never been tested with actual flight parameters
- problem was easily reproduced in simulation, after the fact

the mythical man-month.

- Fred Brooks (1975)
- book written after his experiences in the OS/360 design
- major themes:
  - Brooks’ Law: “Adding manpower to a late software project makes it later.”
  - the “black hole” of large project design: getting stuck and getting out
  - organizing large team projects and communication
  - documentation!!!
  - when to keep code; when to throw code away
  - dealing with limited machine resources
- most are supplemented with practical experience

no silver bullet.

- paper written in 1986 (Brooks)
- “There is no single development, in either technology or management technique, which by itself promises even one order-of magnitude improvement within a decade of productivity, in reliability, in simplicity.”
- why? software is inherently complex
- lots of people disagree(d), but there is no proof of a counter-argument
- Brooks’ point: there is no *revolution*, but there is *evolution* when it comes to software development

mechanics.

- well-established techniques and methodologies:
  - team structures
  - software lifecycle / waterfall model
  - cost and complexity planning / estimation
  - reusability, portability, interoperability, scalability
  - UML, design patterns

team structures.

- why Brooks' Law?
  - training time
  - increased communications: pairs grow by n^2 while people/work grows by n
  - how to divide software? this is *not* task sharing
- types of teams
  - democratic
  - "chief programmer"
  - synchronize-and-stabilize teams
  - eXtreme Programming teams

---

lifecycles.

- software is not a build-one-and-throw-away process
- that's far too expensive
- so software has a *lifecycle*
- we need to implement a *process* so that software is maintained correctly
- examples:
  - build-and-fix
  - waterfall

---

software lifecycle model.

- 7 basic phases (Schach):
  - requirements (2%)
  - specification/analysis (5%)
  - design (6%)
  - implementation (module coding and testing) (12%)
  - integration (8%)
  - maintenance (67%)
  - retirement
- percentages in ()'s are average cost of each task during 1976-1981
- testing and documention should occur throughout each phase
- note which is the most expensive!

---

requirements phase.

- what are we doing, and why?
- need to determine what the client needs, not what the client *wants* or *thinks* they need
- worse — requirements are a moving target!
- common ways of building requirements include:
  - prototyping
  - natural-language requirements document
- use interviews to get information (not easy!)
- example: your online store

specification phase.

- the “contract” — frequently a legal document
- what the product will do, not how to do it
- should NOT be:
  - ambiguous, e.g., “optimal”
  - incomplete, e.g., omitting modules
  - contradictory
- detailed, to allow cost and duration estimation
- classical vs object-oriented (OO) specification
  - classical: flow chart, data-flow diagram
  - object-oriented: UML
- example: your online store

design phase.

- the “how” of the project
- fills in the underlying aspects of the specification
- design decisions last a long time!
- even after the finished product
  - maintenance documentation
  - try to leave it open-ended
- architectural design: decompose project into modules
- detailed design: each module (data structures, algorithms)
- UML can also be useful for design
- example: your online store

implementation phase.

- implement the design in programming language(s)
- observe standardized programming mechanisms
- testing: code review, unit testing
- documentation: commented code, test cases
- integration considerations
  - combine modules and check the whole product
  - top-down vs bottom-up ?
  - testing: product and acceptance testing; code review
  - documentation: commented code, test cases
  - done continually with implementation (can’t wait until the last minute!)
- example: your online store

maintenance phase.

- defined by Schach as *any change*
- by far the most expensive phase
- poor (or lost) documentation often makes the situation even worse
- programmers hate it
- several types:
  - corrective (bugs)
  - perfective (additions to improve)
  - adaptive (system or other underlying changes)
- testing maintenance: regression testing (will it still work now that I’ve fixed it?)
- documentation: record all the changes made and why, as well as new test cases
- example: your on-line store — how might the system change once it’s been implemented?

retirement phase.

- the last phase, of course
- why retire?
  - changes too drastic (e.g., redesign)
  - too many dependencies ("house of cards")
  - no documentation
  - hardware obsolete
- true retirement rate: product no longer useful

planning and estimation.

- we still need to deal with the bottom line
  - how much will it cost?
  - can you stick to your estimate?
  - how long will it take?
  - can you stick to your estimate?
- how do you measure the product (size, complexity)?
-

reusability.

- impediments:
- lack of trust
- logistics of reuse
- loss of knowledge base
- mismatch of features

reusability: how to.

- libraries
- APIs
- system calls
- objects (OOP)
- frameworks (a generic body into which you add your particular code)

portability.

- Java and C#
- Java: uses a JVM
  - write once, run anywhere (sorta, kinda)
- C#: also uses a JVM
  - emphasizes mobile *data* rather than code
- winner?
  - betting against Microsoft is historically a losing proposition...

interoperabilty.

- e.g., CORBA
- define abstract services
- allow programs in any language to access services in any language in any location
- object-ish

scalability.

- something to keep in mind
- don’t worry about scaling beyond the abilities of the machine
- avoid unnecessary barriers
- from single connection to forking processes to threads...