Fall 2004 Handout 6

Give birth to a child: David Curry's O'Reilly book *Using C on the UNIX System* **pp. 292−295; KP pp. 184−185; K&R pp. 167, 253**

The string that you give to **`system`** must use the Bourne shell syntax. The standard output of the **`cal`** in line 7 will become part of the standard output of the following C program:

—Source code on the Web at **`http://i5.nyu.edu/˜mm64/x52.9544/src/system.c`**

```
 1 #include <stdio.h>
 2 #include <stdlib.h>  /* for system */
 3
 4 int main()
 5 {
 6     printf("The current month is\n");
 7     fflush(stdout);
 8
 9     system("cal");   /* Don't need newline. */
10     system("cal 12 2000");
11     system("cal 12 2000 > $HOME/cal.out");    /* Bourne shell won't take tilde. */
12
13     system("who | wc -l");
14     system("grep can\\'t /lyrics/stones/satisfaction");
15     system("grep \"can't\" /lyrics/stones/satisfaction");
16
17     return EXIT_SUCCESS;
18 }
```

```
    1$ grep can\'t /lyrics/stones/satisfaction
    2$ grep "can't" /lyrics/stones/satisfaction
```
KP p. 75

To get the exit status of the program run by **`system`**, store the return value in an **`int`** variable and examine it with the **`W`** macros in **`wait`**(2). Not every process returns an exit status: some are terminated or stopped by a signal first.

—Source code on the Web at **`http://i5.nyu.edu/˜mm64/x52.9544/src/systemexit.c`**

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <sys/wait.h>
 4
 5 int main(int argc, char **argv)
 6 {
 7     /* blank and tab within the [] */
 8     int status = system("who | grep -q '^abc1234[ \t]'");
 9
10     if (WIFEXITED(status)) {
11         printf("My child's exit status was %d.\n", WEXITSTATUS(status));
12     } else if (WIFSIGNALED(status)) {
13         printf("My child was terminated by signal number %d.\n", WTERMSIG(status));
```

```
    } else if (WIFSTOPPED(status)) {
        printf("My child was stopped by signal number %d.\n", WSTOPSIG(status));
    } else {
        fprintf(stderr, "%s: couldn’t find out how child ended up.\n", argv[0]);
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

```
My child’s exit status was 1.
```

system in perl

http://i5.nyu.edu/˜mm64/x52.9544/src/systemexit

```
#!/bin/perl
use POSIX;

$status = system(’who | grep -q \’ˆabc1234[ \t]\’’);

if (WIFEXITED($status)) {
    print "My child’s exit status was ", WEXITSTATUS($status), ".\n";
} elsif (WIFSIGNALED($status)) {
    print "My child was terminated by signal number ", WTERMSIG($status), ".\n";
} elsif (WIFSTOPPED($status)) {
    print "My child was stopped by signal number ", WSTOPSIG($status), ".\n";
} else {
    die "$0: couldn’t find out how child ended up.";
}

exit 0;
```

▼ Homework 6.1: get the child’s exit status

Call `system` to give birth to a child that produces an exit status but no output. Then print a message determined by the exit status of the child. Let the child be one of the following programs, or a pipeline ending with one of the following programs. Or write your own child in C, C++, Perl, or the shell language.

`1$ mail -e`	*exit status is 0 if you have mail*
`2$ grep -q word file`	*exit status is 0 if* `file` *contains* `word`
`3$ cmp -s file1 file2`	*exit status is 0 if* `file1` *and* `file2` *are identical*
`4$ sort -c file 2> /dev/null`	*exit status is 0 if* `file` *is already sorted*
`5$ gcc -o /dev/null prog.c`	*exit status is 0 if* `prog.c` *has no compilation errors*
`6$ test -f file`	*exit status is 0 if* `file` *exists*
`7$ test -f file -a -w file`	*exit status is 0 if* `file` *exists and is writable*
`8$ test -d directory`	*exit status is 0 if* `directory` *exists*
`9$ mkdir directory`	
`10$ test ‘who | awk ’{print $1}’ | sort | uniq | wc -l‘ -gt 20`	
`11$ true`	*exit status always 0*
`12$ false`	*exit status always 1*
`13$ /usr/sbin/ping -c 1 acf5.nyu.edu > /dev/null 2>&1`	*exit status is 0 if* `is` *is online*

Here are some machines you can **`ping`**:

`andrew.cmu.edu`	*Carnegie Mellon University*
`www.uquebec.ca`	*Universite du Quebec*
`www.unipi.it`	*Università degli Studi di Pisa*

▲

Give birth to a child with a pipe from the parent to the child

`popen` runs another program and lets you send output to its standard input. **`pclose`** sends an EOF through the pipe and returns the exit status of the program run by **`popen`**. You can store it in an **`int`** variable and examine it with the **`W`** macros:

—Source code on the Web at
`http://i5.nyu.edu/~mm64/x52.9544/src/popen_to_child.c`

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <sys/wait.h>
 4
 5 int main(int argc, char **argv)
 6 {
 7     FILE *lpr = popen("lpr", "w");
 8     int status;
 9
10     if (lpr == NULL) {
11         perror(argv[0]);
12         return EXIT_FAILURE;
13     }
14
15     fprintf(lpr, "hello\n");
16     fprintf(lpr, "goodbye\n");
17
18     status = pclose(lpr);
19
20     if (WIFEXITED(status)) {
21         printf("My child’s exit status was %d.\n", WEXITSTATUS(status));
22     } else if (WIFSIGNALED(status)) {
23         printf("My child was terminated by signal number %d.\n", WTERMSIG(status));
24     } else if (WIFSTOPPED(status)) {
25         printf("My child was stopped by signal number %d.\n", WSTOPSIG(status));
26     } else {
27         fprintf(stderr, "%s: couldn’t find out how child ended up.\n", argv[0]);
28         return EXIT_FAILURE;
29     }
30
31     return EXIT_SUCCESS;
32 }
```

The string that you give to **`popen`** is not limited to one program. You can change line 7 to

```
 7     FILE *lpr = popen("sort | cat -n | pr -l60 | lpr", "w");
```

Use Bourne shell syntax.

Give birth to a child with a pipe from the child to the parent

—Source code on the Web at
http://i5.nyu.edu/˜mm64/x52.9544/src/popen_to_parent.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(int argc, char **argv)
{
    FILE *wc = popen("who | awk '{print $1}' | sort | uniq | wc -l", "r");
    int n;  /* number of people logged in */
    int status;

    if (wc == NULL) {
        perror(argv[0]);
        return EXIT_FAILURE;
    }

    fscanf(wc, "%d", &n);
    printf("There are %d people logged in.\n", n);

    status = pclose(wc);
    if (!WIFEXITED(status) || WEXITSTATUS(status) != EXIT_SUCCESS) {
        fprintf(stderr, "%s: child came to grief somehow.\n", argv[0]);
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

You can call **popen** several times in the same C program. This allows you to have more than one pipe coming into and/or going out of a C program (or a Perl program), which you can't have in a shellscript.

popen in Perl

```
http://i5.nyu.edu/˜mm64/x52.9544/src/popen
#!/bin/perl
use POSIX;

open(WC, 'who | awk \'{print $1}\' | sort | uniq | wc -l |') || die "$0: $!";
open(LPR, '| lpr') || die "$0: $!";

$_ = <WC>;
chomp;
print LPR "There are $_ people logged in.\n";

close WC;
#Should have checked WIFEXITED before calling WEXITSTATUS.
print 'The exit status of the wc -l was ', WEXITSTATUS($?), ".\n";

close LPR;
print 'The exit status of the lpr was ', WEXITSTATUS($?), ".\n";

exit 0;
```

▼ **Homework 6.2: pipe data to sort**

Make Homework 13.4 list everything in alphabetical order. Simply use `popen`, `fprintf`, and `pclose` to pipe your C program's output to `sort +8`.

```
int main(int argc, char **argv)
{
    opendir;
    popen("sort", "w");

    fprintf all the output into the pipe;

    pclose;
    closedir;
}
```

If you have done the extra credit parts of Homework 1.8, some lines of output will not have nine fields, so `sort +8` won't work. Use

```
awk '{print $NF, $0}' | sort | sed 's/^[^ ][^ ]* //'
```

instead. (The `sed` remove everything up to and including the first blank on each line.)

▲

Archive several files into one big .tar file

`tar` is a utility for writing a group of files onto a tape, creating a *tape archive*. But the output of `tar`, like that of any Unix program, can be directed to a file instead of to a hardware device. Give the file a name ending with `.tar`.

The following example could have used a device name such as `/dev/rmtnh` (raw magnetic tape) instead of the filename `date.tar`. In that case you'd also need the `mt rewind` command.

```
1$ cd
2$ date > date1
3$ date > date2
4$ date > date3
5$ ls -l date[1-3]
-rw-------   1 mm64     users         29 Jan  9 00:41 date1
-rw-------   1 mm64     users         29 Jan  9 00:41 date2
-rw-------   1 mm64     users         29 Jan  9 00:41 date3

6$ tar cvf date.tar date1 date2 date3                  create date.tar
a date1 1K
a date2 1K
a date3 1K

7$ ls -l date.tar
-rw-------   1 abc1234  users       4096 Jan  9 00:41 date.tar

8$ tar tvf date.tar | more          Output a table of contents of the .tar file.
tar: blocksize = 8
-rw------- 50766/15      29 Jan  9 00:41 2004 date1
-rw------- 50766/15      29 Jan  9 00:41 2004 date2
-rw------- 50766/15      29 Jan  9 00:41 2004 date3
```

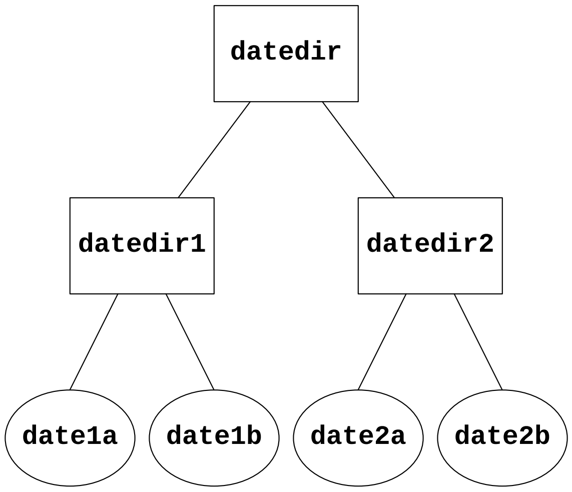
The **x** flag extracts a copy of one or more of the little files archived in the **.tar** file. To extract everything archived in the **.tar** file, simply give no arguments after the name of the **.tar** file. Extraction does not change the contents of the **tar** file.

```
9$ rm date1 date2 date3             or rm date[1-3] p. 28; ksh93(1) pp. 10−11

10$ tar xvf date.tar date1          create date1
tar: blocksize = 8
x date1, 29 bytes, 1 tape blocks

11$ ls -l date1
-rw-------   1 abc1234  users         29 Jan  9 00:41 date1
```

Archive an entire directory into one big .tar file



If you give one or more directory names instead of one or more filenames to **`tar cvf`** after the name of the **`.tar`** file, **`tar`** will archive the directories and all of their descendants, including all of the files they contain.

```
1$ cd
2$ pwd
/home1/a/abc1234

3$ mkdir datedir

4$ mkdir datedir/datedir1
5$ date > datedir/datedir1/date1a
6$ date > datedir/datedir1/date1b

7$ mkdir datedir/datedir2
8$ date > datedir/datedir/date2a
9$ date > datedir/datedir/date2b

10$ tar cvf date.tar datedir
a datedir/ 0K
a datedir/datedir1/ 0K
a datedir/datedir1/date1a 1K
a datedir/datedir1/date1b 1K
a datedir/datedir2/ 0K
a datedir/datedir2/date2a 1K
a datedir/datedir2/date2b 1K
```

```
    11$ tar tvf date.tar | more
    tar: blocksize = 13
    drwx------ 50766/15       0 Jan  9 00:41 2004 datedir/
    drwx------ 50766/15       0 Jan  9 00:41 2004 datedir/datedir1/
    -rw------- 50766/15      29 Jan  9 00:41 2004 datedir/datedir1/date1a
    -rw------- 50766/15      29 Jan  9 00:41 2004 datedir/datedir1/date1b
    drwx------ 50766/15       0 Jan  9 00:41 2004 datedir/datedir2/
    -rw------- 50766/15      29 Jan  9 00:41 2004 datedir/datedir2/date2a
    -rw------- 50766/15      29 Jan  9 00:41 2004 datedir/datedir2/date2b

    12$ rm    datedir/datedir[12]/*
    13$ rmdir datedir/datedir[12]
    14$ rmdir datedir
```

The following **`tar xvf`** command will re-create the directory **`datedir`**, its subdirectory **`datedir2`**, and the file **`date2a`**:

```
    15$ tar xvf date.tar datedir/datedir2/date2a
    tar: blocksize = 13
    x datedir/datedir2/date2a, 29 bytes, 1 tape blocks

    16$ ls -l | more
    drwx------   4 abc1234  users        257 Jan  9 00:41 datedir

    17$ ls -l datedir | more
    drwx------   2 abc1234  users        253 Jan  9 00:41 datedir/datedir2

    18$ ls -l datedir/datedir2/date2a
    -rw-------   1 abc1234  users         29 Jan  9 00:41 datedir/datedir2/date2a
```

How much of the full pathnames should be included in the .tar file?

If you specify the full pathnames of the files and directories to be archived, then their full pathnames will be stored in the **`.tar`** file:

```
    1$ rm date.tar
    2$ tar cvf date.tar ~/datedir
    a /home1/a/abc1234/datedir/ 0K
    a /home1/a/abc1234/datedir/datedir1/ 0K
    a /home1/a/abc1234/datedir/datedir1/date1a 1K
    a /home1/a/abc1234/datedir/datedir1/date1b 1K
    a /home1/a/abc1234/datedir/datedir2/ 0K
    a /home1/a/abc1234/datedir/datedir2/date2b 1K
    a /home1/a/abc1234/datedir/datedir2/date2a 1K

3$ tar tvf date.tar
tar: blocksize = 13
drwx------ 50766/15       0 Jan  9 00:41 2004 /home1/a/abc1234/datedir/
drwx------ 50766/15       0 Jan  9 00:41 2004 /home1/a/abc1234/datedir/datedir1/
-rw------- 50766/15      29 Jan  9 00:41 2004 /home1/a/abc1234/datedir/datedir1/date1a
-rw------- 50766/15      29 Jan  9 00:41 2004 /home1/a/abc1234/datedir/datedir1/date1b
drwx------ 50766/15       0 Jan  9 00:41 2004 /home1/a/abc1234/datedir/datedir2/
-rw------- 50766/15      29 Jan  9 00:41 2004 /home1/a/abc1234/datedir/datedir2/date2b
-rw------- 50766/15      29 Jan  9 00:41 2004 /home1/a/abc1234/datedir/datedir2/date2a
```

If you specify only the basenames of the files and directories to be archived, then only their basenames will be stored in the `.tar` file:

```
4$ rm date.tar
5$ cd ~/datedir/datedir1
6$ tar cvf ~/date.tar date1a date1b
a date1a 1K
a date1b 1K

7$ cd ~/datedir/datedir2
8$ tar rvf ~/date.tar date2a date2b
a date2a 1K
a date2b 1K

9$ tar tvf ~/date.tar
tar: blocksize = 10
-rw------- 50766/15      29 Jan  9 00:41 2004 date1a
-rw------- 50766/15      29 Jan  9 00:41 2004 date1b
-rw------- 50766/15      29 Jan  9 00:41 2004 date2a
-rw------- 50766/15      29 Jan  9 00:41 2004 date2b
```

"replace", not **`cvf`***!*

Another way to do exactly the same thing is to use the uppercase **`-C`** option of **`tar`** instead of doing the **`cd`**'ing yourself:

```
10$ cd
11$ rm date.tar
12$ tar cvf date.tar \
    -C datedir/datedir1 date1a \
    -C datedir/datedir1 date1b \
    -C datedir/datedir2 date2a \
    -C datedir/datedir2 date2b
a date1a 1K
a date1b 1K
a date2a 1K
a date2b 1K

13$ pwd
/home1/a/abc1234
```

The **`-C`** *effected only the* **`tar`***, not you.*

```
14$ tar tvf date.tar
tar: blocksize = 10
-rw------- 50766/15      29 Jan  9 00:41 2004 date1a
-rw------- 50766/15      29 Jan  9 00:41 2004 date1b
-rw------- 50766/15      29 Jan  9 00:41 2004 date2a
-rw------- 50766/15      29 Jan  9 00:41 2004 date2b
```

Exclude one or more files and/or subdirectories

To archive the directory **`datedir`** and all of its descendants except for the directory **`datedir/datedir1`**, use the **`-e`** option. Specify the exception directory **`datedir/datedir1`** *before* its ancestor directory **`datedir`**:

```
1$ cd
2$ tar cvf date.tar -e datedir/datedir1 datedir
```

```
    3$ tar tvf date.tar
```

▼ **Homework 6.3: create a tar file**

Verify that all of the above works. If you're allowed to use a tape drive, **`tar`** some files to tape instead of to a **`.tar`** file.

▲) .HW tar the ppm.h file and all the .c files and the directories that contain them

Create a file named **`ppm.tar`** containing the **`$m46/ppm`** directory and all of its descendants except for the subdirectories **`$m46/ppm/bin`** and **`$m46/ppm/lib`**.

Since "verbose" output of **`tar cvf`** is directed to the standard error output rather than to the standard output, you'll have to use **`|&`** rather than **`|`** if you want to pipe it to **`more`**. See p. 13 of **`csh`**(1). (The output of **`tar tvf`** goes to the standard output.)

Hand in a **`tar tvf`** of your **`ppm.tar`** file. It must contain only the part of the pathnames from **`ppm`** onwards:

```
1$ tar tvf ppm.tar | more
```

▲

▼ **Homework 6.4: copy all your files from one machine to another**

```
    1$ cd
    2$ pwd

    3$ tar cvf all.tar .
    4$ tar tvf all.tar | more
    5$ tar tvf all.tar | lpr
```

Use **`ftp`** to copy the above **`binary`** file named **`all.tar`** to your home directory on another machine. Then on the other machine,

```
    $ cd
    $ ls -l all.tar
    $ tar tvf all.tar | more
    $ tar tvf all.tar | lpr
    $ tar xvpf all.tar                          p because machines have different umask
```

▲

A comparison of ar and tar

`ar` usually creates a file containing an library of **`.o`** files. It also lets you insert new members at any point in an existing library, change the order of existing members, and delete members.

`tar` often sends its output to a tape instead of a file. For this reason, it lets you add new members only to the end of the archive, and has no facilities for reordering or deleting members.

`ar` creates an index (_________**`64ELEL`**_), **`tar`** doesn't. In place of the tape drive **`/dev/rmt0h`** below ("raw magnetic tape"), you can use a file whose name ends with **`.tar`**.

Create an archive with three members:

```
    1$ ar crsv libdate.a  date1.o date2.o date3.o
    2$ tar cvf /dev/rmt0h date1.c date2.c date3.c
```

Output the table of contents of an archive:

```
    3$ ar tv libdate.a
    4$ mt -f /dev/rmt0h rewind
    5$ tar tvf /dev/rmt0h
```

Create a copy of **date2.o** *or* **date2.c** *in the current directory.*

```
6$ ar xv libdate.a date2.o
7$ tar xvf /dev/rmt0h date2.c
```

Compress and uncompress

```
1$ cd
2$ cp /etc/passwd .                        Copy /etc/passwd to your current directory.
3$ ls -l passwd
-r--------   1 abc1234  users     736431 Jan  9 00:41 passwd

4$ compress passwd                         remove passwd and create passwd.Z
5$ ls -l passwd.Z
-r--------   1 abc1234  users     221463 Jan  9 00:41 passwd.Z
```

Do not **cat** to the screen or **lpr** a compressed file (in this case, a **.Z** file). Instead,

```
6$ zcat passwd.Z | head -3
root:x:0:1:Super-User:/:/sbin/sh
daemon:x:1:1::/:
bin:x:2:2::/usr/bin:

7$ uncompress passwd.Z                     remove passwd.Z and create passwd
8$ ls -l passwd
-r--------   1 abc1234  users     736431 Jan  9 00:41 passwd
```

The compression and decompression is not “lossy”:

```
9$ cmp passwd /etc/passwd                  No output if identical: Handout 2, p. 11.
10$ rm passwd
```

Other compression programs

suffix	*compress*	*decompress*
.Z	**compress**	**uncompress**
.gz	**gzip**	**gunzip**
.z	**pack**	**unpack**

Programs written by GNU often start with **g**. See **http://www.gnu.org/**. To list the pairs of utilities for compression and decompression on i5.nyu.edu,

```
1$ man -k compress | more                  search for a keyword with -k
```

For a comprehensive list of suffixes, pairs of utilities, and how to get them, see David Lemson’s compression document,

http://www.lemson.com/lemson/work.html

A shorter list is in the *Whole Internet Catalog, 2nd ed.*, pp. 83−85.

▼ Homework 6.5: compress and uncompress a file

Compress and uncompress a file (ASCII or binary). Use **bc** to find the percent by which it was compressed. Which pair of utilities yields the most compression? Do text files compress further than image files?

```
1$ bc
scale = 2
100 * 221463 / 736431
30.07
control-d
2$
```

Handout 2, p. 21; Handout 4, pp. 23−24; Handout 7, p. 7
Output answers to two decimal places.

▲ Compress ppm.tar

Compress the **`ppm.tar`** file that you created in a previous homework, creating a file named **`ppm.tar.Z`**. Hand in the output of

```
zcat ppm.tar.Z | tar tvf -
```

uuencode and uudecode

`mail` and **`nn`** can send only text files, not binaries.

```
/* This file is little.c. */
main ()
{
}
```

```
1$ cc -o little little.c
2$ ls -l little
-rwx------   1 mm64     users       5916 Jan  9 00:41 little

3$ uuencode little little2 > ascii
4$ head -5 ascii
begin 700 little2
M?T5,1@$" 0             "  (    !  $$!    #0  !+D       T "
M!0 H !L &0    8    T  $ -         "@    H     4          P
M -0              !$         !          !       !           &
M/   !CP    %  $       $   8  (&/         %D   !@     <  0

5$ tail -1 ascii
end

6$ rm little
7$ uudecode ascii
8$ ls -l
-rwx------   1 mm64     users       5916 Jan  9 00:41 little2
```

▼ **Homework 6.6: the Revision Control System RCS**

Please change every **`46`** to **`45`** in the RCS section. For example, **`~mm64/46`** becomes **`~mm6445`**.

The Concurrent Versions System CVS sits on top of RCS. An older program similar to RCS was the Source Code Control System SCCS.

Do not attempt this assignment before I discuss it in class, or if you were absent when I discussed it in class, or if your **`vi`** says **`[Using open mode]`** (Handout 3, p. 1), or if **`echo $S45`** shows you nothing (Handout 2, p. 13, lines 34−35). Do not write a lawyer joke.

Do not edit any file in the **`~mm64/46/RCS`** directory. Do not move or copy any file into the **`~mm64/46/RCS`** directory, or remove any file from that directory.

Do everything in your home directory. Change the permission bits of your home directory to **`rwxr-xr-x`** if you have not already done so.

(1) The **co** command (“check out”) should make the **jokes** file appear in your current directory. You can do anything you want with this copy of **jokes**: **more** it, **lpr** it, **vi** it, etc. But the **co** command does not authorize you to change the master copy of **jokes** from which your copy was cloned. **co** therefore entails no responsibility: it doesn’t hold up anyone else’s work.

You can also see the **jokes** file, the person (if any) who is editing it now, and a list of all the people who have edited it on the X52.9545 home page on the World Wide Web at
http://i5.nyu.edu/~mm64/x52.9545#jokes

```
1$ cd
2$ pwd

3$ ls -ld                 Make sure your home directory is rwxr-xr-x.
4$ ls -l jokes            If a jokes file already exists, move or remove or rename it.

5$ co ~mm64/46/RCS/jokes
/home1/m/mm64/46/RCS/jokes  -->  jokes
revision 1.3
done

6$ ls -l jokes            There’s a new file named jokes.
-r--------   1 abc1234  users        512 Nov  2 21:00 jokes
7$ more jokes
8$ rm -f jokes            -f to remove a file with no w permission: p. 56
```

(2) The **co -l** command also makes the **jokes** file appear in your current directory. The **-l** option authorizes you to use the **ci** command in paragraph (3) to change the master copy of **jokes** from which your copy was cloned. You can then edit **jokes** with any editor:

```
9$ cd
10$ pwd

11$ ls -ld                Make sure your home directory is rwxr-xr-x.
12$ ls -l jokes           If a jokes file already exists, move or remove or rename it.

13$ co -l ~mm64/46/RCS/jokes                     minus lowercase L for “lock”
/home1/m/mm64/46/RCS/jokes  -->  jokes
revision 1.3 (locked)
done

14$ ls -l jokes           There’s a new file named jokes.  Only you can edit it.
-rw-------   1 abc1234  users        512 Nov  2 21:00 jokes
15$ vi jokes
```

(3) When you have finished typing your joke at the bottom of the **jokes** file, use the **ci** command (“check in”) to put the file back so that other people can have their turn. **ci** will ask you to describe what you did to **jokes**. Tell **ci** which joke(s) you added, pressing **RETURN** at the end of each line. After the **RETURN** at the end of the last line, type **control-d** (Handout 2, p. 21). **ci** will then make **jokes** disappear from your current directory; type **ls -l** to make sure that it’s gone. You are now done.

```
16$ ci ~mm64/46/RCS/jokes                         Don't forget the ~mm64/46/RCS/
/home1/m/mm64/46/RCS/jokes  <--  jokes                              It says this.
new revision: 1.4; previous revision: 1.3
enter log message, terminate with single '.' or end of file:
>> I added the jokes about the chicken, snowman, and route 9W. RETURN
>> control-d
done

17$ ls -l jokes                     The file jokes is gone.
18$ rlog -L -R ~mm64/46/RCS/jokes   No output means you no longer have permission to edit jokes.
```

If **rlog -L -R ~mm64/46/RCS/jokes** does not remain silent, it means that your **ci** command didn't work. **jokes** is still being edited, and you haven't relinquished permission to edit the file. No one else can type a joke. Give the **rcs -o** command shown below (minus lowercase o: "outdate" means delete), follow it up with another **rlog -L -t**, and send me **mail** immediately with your login name and secret password:

```
19$ mail mark.meretzky@nyu.edu
Subject: ci failed
I gave the command
ci ~mm64/46/RCS/jokes
and typed my description and pressed control-d, but the command
rlog -L -t ~mm64/46/RCS/jokes
says that I'm still editing the jokes file.
My login name is abc1234 and my secret password is Bacall8?.
control-d
20$
```

(4) After you have successfully **ci**'d, give the **co** command without the **-l** option and print out the **jokes** file. ★ You get credit only if you print the copy of the file obtained from **co** without the **-l** option. The copy obtained from **co** without the **-l** will have the revision number after the **$Header$** on the first line; the one obtained from **co** with the **-l** will have no revision number after the **$Header$**. Draw a circle around your joke and hand in the printout of the entire **jokes** file. You get credit only if you draw a circle around your joke.

```
21$ cd
22$ pwd

23$ ls -l jokes                If a jokes file already exists, move or remove or rename it.
24$ co ~mm64/46/RCS/jokes
25$ pr -l60 jokes | lpr        minus lowercase L sixty; hand in this printout
26$ rm jokes
```

If your joke does not print or is mangled (e.g., if you typed lines that are longer than 80 characters), get permission to edit the file again, fix your joke, and **ci** it again. Then **co** the file again, without the **-l**, and see if your joke prints correctly.

What can go wrong in RCS

(1) If the **co -l ~mm64/46/RCS/jokes** command says

```
co error: revision 1.3 already locked by abc1234
```

it means that **abc1234** is editing the file. Wait your turn or pester him or her (see below). To see a list of all the files that RCS has given permission to edit,

```
1$ rlog -L -R ~mm64/46/RCS/*
```

(2) If the **`co -l ~mm64/46/RCS/jokes`** command complains

```
writable jokes exists; remove it? [ny](n):
```
you should say **`n`**

it means that there already was a file named **`jokes`** in your current directory, and **`co -l`** refused to overwrite it. **`rm`** the file and try again.

(3) If the **`co`** command (with or without the **`-l`** option) complains

```
co error: /home1/m/mm64/45/RCS/jokes: Permission denied
```

it means that the instructor forgot the **`chmod 444`** in Handout 6, p. 4. Please send him email reminding him. Thanks.

(4) If you realize while you're editing that you are only making the file worse, exit from **`vi`** and cancel the **`co -l`** with the **`-u`** option ("unlock") of **`rcs`**:

```
2$ rcs -u ~mm64/46/RCS/jokes
RCS file: /home1/m/mm64/46/RCS/jokes
1.3 unlocked
done
```

(5) If you realize after you've **`ci`**'d that your new revision (number 1.4) was worse than the old revision, you can remove the new revision by immediately saying

```
3$ rcs -o1.4 ~mm64/46/RCS/jokes
```
minus lowercase O stands for "outdate"

Other RCS commands

(1) To see a list of all the people who have edited **`jokes`**, and the revision numbers,

```
1$ rlog ~mm64/46/RCS/jokes | more
```

If **`jokes`** is being edited, you will see the login name of the person who is editing it on the line below the one that says **`locks:`**.

(2) By default, the **`co ~mm64/46/RCS/jokes`** command gets the most recent revision of the file. To get a previous revision, use the **`-r`** option:

```
2$ co -r1.2 ~mm64/46/RCS/jokes
```

(3) The RCS documentation is under **`rcsintro`**(1), **`co`**(1), **`ci`**(1), **`rlog`**(1), **`rcs`**(1), **`ident`**(1).

How I created the jokes file

To create an empty file named **`jokes`** and put it under the protection of RCS, I first created a subdirectory of **`~mm64/46`** named **`RCS`** and **`chmod`**'ed it. Never edit any file in an **`RCS`** directory, and never **`co`** or **`ci`** while you're in an **`RCS`** directory.

```
1$ cd ~mm64/46
2$ pwd

3$ mkdir RCS                                 must be uppercase
4$ ls -ld RCS
drwx------   2 mm64     users    512 Nov  2 12:41 RCS

5$ chmod 777 RCS
6$ ls -ld RCS
drwxrwxrwx   2 mm64     users    512 Nov  2 12:41 RCS

7$ rcs -i '-t-X52.9546/Y12.1005 Fall 2004 jokes' ~mm64/46/RCS/jokes
8$ rlog ~mm64/46/RCS/jokes               Make sure that the rcs -i -t- worked.
```

```
9$ chmod 444 ˜mm64/46/RCS/jokes                  Turn on all three r bits.
10$ ls -l ˜mm64/46/RCS/jokes              Make sure that the chmod worked.
```

To put an existing file under the protection of RCS, it's more convenient to use the **ci -t- ˜mm64/46/RCS/jokes** command instead of the **rcs -i -t- ˜mm64/46/RCS/jokes** shown above.

Insert the revision number into your file

If you type the expression

```
$Header$
```

into a file under the protection of RCS, the expression will be changed to

```
"$Header: /home1/m/mm64/46/RCS/jokes 1.4 2004/01/13 18:11:07 abc1234 Exp $";
```

when you check it out with **co** without the **-l** option. See the first line of the **jokes** file. See **ident**(1) and p. 3 of **rcsintro**(1).

Make bibliography

See the three digressions on **make** in the textbook on pp. 241−242, 254−256, and 265−266. Page numbers below refer to the 9-page manual page **make**(1). See also **make**(1p) (Posix) and **make**(1u) (Ultrix). Also print (with minus lowercase L sixty)

```
1$ make -p -f /dev/null | pr -l60 -h 'make internal rules' | lpr
```

See also *Managing Projects with make, 2nd ed.*, by Andrew Oram and Steve Talbott; O'Reilly & Associates, 1991; ISBN 0-937175-90-0; **http://www.oreilly.com/catalog/make2/**

Sample C program to demonstrate make

```
/* This file is func.h. */
int f(void);
```

```
/* This file is var.h. */
extern int i;
```

```
/* This file is main.c. */
#include <stdio.h>
#include <stdlib.h>
#include "var.h"
#include "func.h"

int main(int argc, char **argv)
{
    printf("%d\n", i + f());
    return EXIT_SUCCESS;
}
```

```
/* This file is func.c. */
#include "func.h"

int f(void)
{
    return 2;
}
```

```
/* This file is var.c. */
#include "var.h"

int i = 1;
```

```
#!/bin/sh
#Compile and link the above C program.

gcc -c main.c                           #create main.o
gcc -c func.c                           #create func.o
gcc -c var.c                            #create var.o
gcc -o prog main.o func.o var.o         #create prog
```

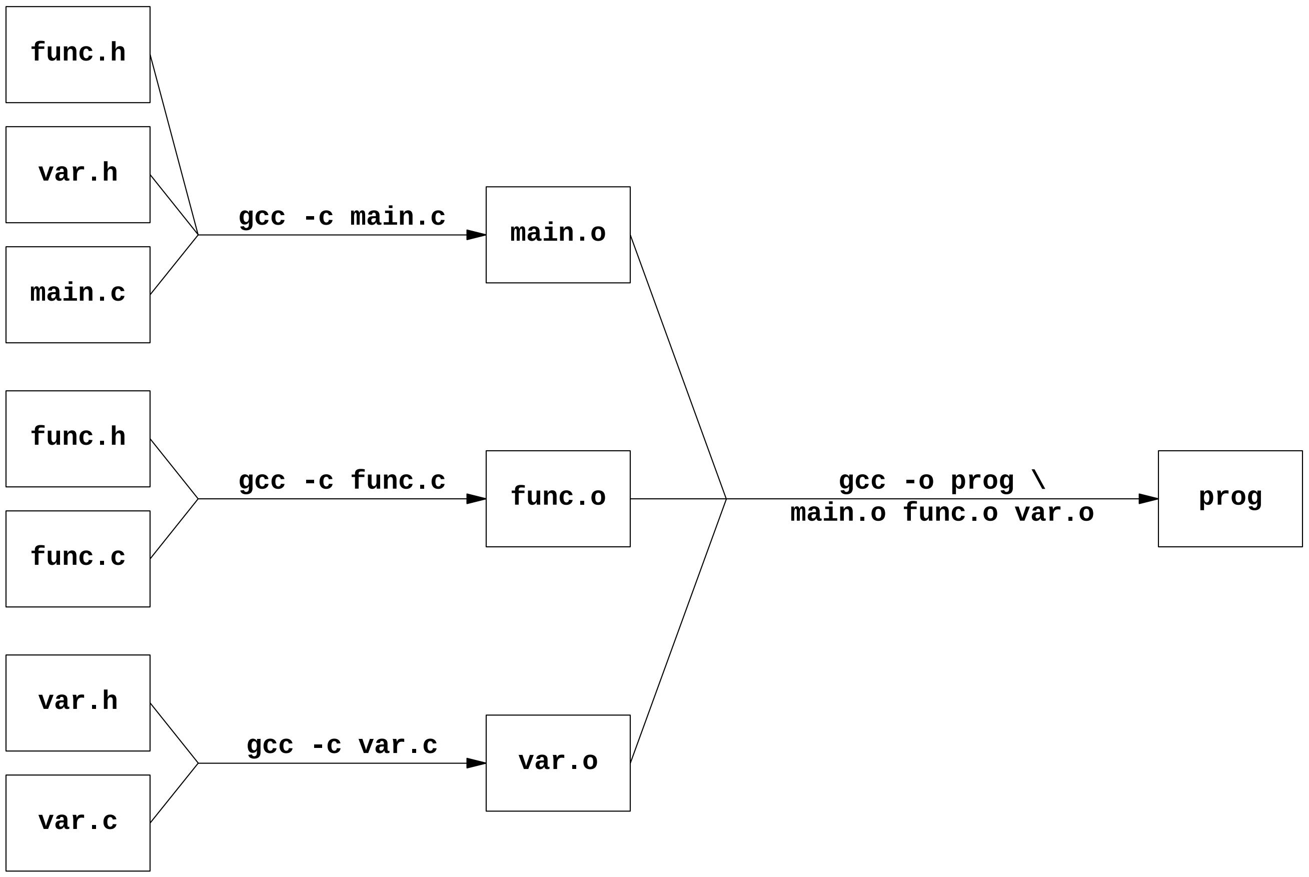
Which files need to be recreated?

A **.o** file needs to be recompiled if it is older than the corresponding **.c** file or any of the **.h** files that it **#include**'s. An executable file needs to be relinked if it is older than any of the **.o** files it comprises.

```
1$ cd $m46/make
2$ ls -l | tail +2
-rw-r--r--   1 mm64     users     73 Nov  2 13:21 func.c
-rw-r--r--   1 mm64     users     40 Nov  2 13:24 func.h
-rw-r--r--   1 mm64     users    496 Nov  2 13:25 func.o
-rw-r--r--   1 mm64     users    136 Nov  2 13:22 main.c
-rw-r--r--   1 mm64     users   1764 Nov  2 13:25 main.o
-rwxr-xr-x   1 mm64     users  40688 Nov  2 13:26 prog
-rw-r--r--   1 mm64     users     55 Nov  2 13:27 var.c
-rw-r--r--   1 mm64     users     40 Nov  2 13:24 var.h
-rw-r--r--   1 mm64     users    428 Nov  2 13:25 var.o

3$ ls -lt | tail +2
-rw-r--r--   1 mm64      users    55 Nov  2 13:27 var.c
-rwxr-xr-x   1 mm64      users 40688 Nov  2 13:26 prog
-rw-r--r--   1 mm64      users   428 Nov  2 13:25 var.o
-rw-r--r--   1 mm64      users  1764 Nov  2 13:25 main.o
-rw-r--r--   1 mm64      users   496 Nov  2 13:25 func.o
-rw-r--r--   1 mm64      users    40 Nov  2 13:24 var.h
-rw-r--r--   1 mm64      users    40 Nov  2 13:24 func.h
-rw-r--r--   1 mm64      users   136 Nov  2 13:22 main.c
-rw-r--r--   1 mm64      users    73 Nov  2 13:21 func.c
```

The dependency tree



A simple makefile

Put all of the **.c** and **.h** files of your C program (except for the **.h** files in the **/usr/include** directory) into one directory, together with the file named **makefile** shown below. Since **makefile** is not a shellscript, do not start it with **#!** or turn on its three **x** bits.

The file before the colon is the *target*. The files after the colon are the *dependents*. A line with a colon and the line(s) indented below it constitute a *rule*. Do not indent the line with the target and dependency files, but indent the line(s) below them that tell how to create the target file. Skip an empty line between rules.

In **makefile**, all indentation must be by EXACTLY ONE TAB CHARACTER. Do not indent with blanks. This is the only place in Unix where the difference between blanks and tabs is significant. The *Unix Haters Handbook* (by Simon Garfinkel, Daniel Weise, and Steven Strassmann, with a foreward by Dennis Ritchie; IDG Books, 1994; ISBN 1-56884-203-1), p. 185, says ‘‘According to legend, Stu Feldman [the creator of **make**] didn’t fix **make**’s syntax, after he realized that the syntax was broken, because he already had 10 users.’’

The following **makefile** contains four rules. A rule is executed if the target doesn’t exist, or if the target is older than any of its dependents, or if there are no dependents listed to the right of the colon.

```
prog: main.o var.o func.o
    cc -o prog main.o var.o func.o

main.o: main.c var.h func.h
    cc -c main.c

var.o: var.c var.h
    cc -c var.c

func.o: func.c func.h
    cc -c func.c
```

You can create any of the targets in the **makefile**. **make** will do all the compiling and linking that is necessary, and no more.

```
1$ cd  to the directory that contains the .h and .c files and the makefile
2$ make prog                     create prog
3$ make main.o                   create main.o
4$ make var.o                    create var.o
5$ make func.o                   create func.o
```

If you give **make** no command line argument, you will create the first target in the **makefile** by default. That's why the target at the root of the tree is listed first in the **makefile**. The other targets can be listed in any order.

```
6$ make                          create prog
7$ make                          Nothing happens the second time.
'prog' is up to date.
```

▼ Homework 6.7: play with make

The files **func.h**, **var.h**, **main.c**, **func.c**, **var.c**, and **makefile** are in the directory **$m46/make**. Copy them to a directory named **$HOME/prog**. Then

```
1$ cd $HOME/prog
2$ make
3$ ls -l                Look at the new files created by the make command.
4$ prog                   Run the program created by the make command.
```

Then run the **make** command again and verify that no additional compilation or linking takes place.

Now edit one of the **.c** files, and verify that **make** compiles only the edited file and then relinks the executable. Then edit one of the **.h** files and verify that **make** compiles every **.c** file that **#include**'s the **.h** file (but no other **.c** file) and then relinks the executable.

Instead of editing a **.c** file, give the **touch** command to let you experiment more rapidly.

```
5$ ls -l main.c
-rw-r--r--   1 abc1234  users        136 Nov  2 13:22 main.c

6$ touch main.c                         faster than vi main.c
7$ ls -l
-rw-r--r--   1 abc1234  users        136 Nov  2 13:29 main.c
```

You can confuse **make** by saying

```
8$ vi main.c                     This would normally cause make to recompile main.c.
9$ touch main.o
10$ touch prog
11$ make                                            Does make recompile main.c?
```

{ } around a shell variable: p. 148

```
1$ lpq -Pth_hp4si_1
2$ lpq -Ped_hp4si_1
```

```
#!/bin/sh
#Print the queue for each Hewlett Packard 4Si laser printer.
#Without the {}, the echo would print the wrong variable.

for p in th ed
do
    echo ${p}_hp4si_1:
    lpq -P${p}_hp4si_1:
done

exit 0
```

Put parentheses or curly braces around the name of every **make** macro if the name is more than one character long.

A makefile with macros

$@** and **$* are internal macros, i.e., variables to which **make** gives different values automatically in each rule where they are used. See p. 5 in **make**(1). **$@** is the name of the target file, and **$*** is the name of the target file with the suffix removed (i.e., the basename of the target file). Contrary to what the **man** says, **$*** can be used outside of suffix rules. You can use **$*** only in an indented line, not in a colon line.

CC, **CFLAGS**, and **OBJS** are non-internal macros; see p. 255 in the textbook and pp. 3−5 in **make**(1). A non-internal macro can have any name you want, but please pick names that agree with those output by the **make -p** command above.

```
#makefile for the above C program, using macros.

CC = gcc
CFLAGS = -O      #optimization
OBJS = main.o func.o var.o

prog: $(OBJS)
    $(CC) $(CFLAGS) -o $@ $(OBJS)

main.o: main.c func.h var.h
    $(CC) $(CFLAGS) -c $*.c

func.o: func.c func.h
    $(CC) $(CFLAGS) -c $*.c

var.o: var.c var.h
    $(CC) $(CFLAGS) -c $*.c
```

▼ Homework 6.8: create a makefile for moon

```
1$ cd
2$ mkdir moon
3$ cd moon
4$ cp $m46/moon/moon*.[ch] .                    Handout 1, p. 14
5$ chmod 644 moon*.[ch]                  if you plan to edit these files
```

Create a **`makefile`** for **`moon`** in your **`$HOME/moon`** directory. Hand in the **`makefile`**. Use the internal macros **`$@`** and **`$*`**. Create the macros **`CC`**, **`CFLAGS`**, and **`OBJS`**. Leave the **`CFLAGS`** macro empty if you don't want optimization:

```
CFLAGS =
```

Also create a macro named **`LOADLIBES`** to hold the **`-lm`** option of **`gcc`** that **`moon`** requires. Write **`$(LOADLIBES)`** at the end of the line that contains **`-o $@`**.
▲

Other ways to create a make macro: p. 3.

Instead of defining a macro in your **`makefile,`** you can pass a command line argument to **`make`**:

```
1$ make CC=gcc                           No space around the equal sign.
```

or set an environment variable:

```
2$ setenv CC gcc          do this in your .login file.
3$ env | more             see the names and values of all your environment variables
4$ make
```

If you a macro without defining it anywhere, **`make`** will use the default definition you printed out with the **`make -p`** command. These rules also include the defaults for any rules you left out of your **`makefile`**. For example, if you don't say how to create a **`.o`** file from a **`.c`** file, **`make`** will use the **`.c.o`** default rule displayed by the **`make -p`** command.

A makefile for an archive

An archive is made out of **`.o`** files, just as a **`.o`** file is made out of a **`.c`** file. The **`.o`** files inside of the archive **`libppm.a`** are named **`libppm.a(ppm_inheader.o)`**, **`libppm.a(ppm_outheader.o)`**, etc. Use backslashes to divide a long colon statement into separate lines.

The internal macro **`$?`** in the following **`makefile`** holds the names of all the **`.o`** files in the library that need to be recompiled (i.e., that are older than the corresponding **`.c`** files); see p. 3. The macro **`$(?:.o=.c)`** is **`$?`** with the **`.o`**'s at the end of each word changed to **`.c`**'s; see p. 4.

Each indented line counts as a separate shellscript. To write a multi-line command such as **`if-then-else-fi`**, you must therefore use backslashes.

```
#!/bin/sh
#What goes wrong if you omit the semicolon?

grep word file
who

grep word file; who
```

```
#!/bin/sh
#What goes wrong if you omit the semicolons?

if grep -q word file
then
    who
fi

if grep -q word file; then who; fi
```

```
#This file is $m46/ppm/src/makefile.

CFLAGS = -I/home/m/mm64/46/ppm/include

libppm.a: \
    libppm.a(ppm_inheader.o) \
    libppm.a(ppm_outheader.o) \
    libppm.a(ppm_negative.o)
    $(CC) $(CFLAGS) -c $(?:.o=.c)
    if [ -f $@ ];\
    then\
        ar rsv $@ $?;\
    else\
        ar crsv $@ $?;\
    fi
    rm $?

#Disable the default rule for creating a .a file out of .c files to
#allow the above rule to be used instead.
.c.a:;
```

A taller tree

The above tree diagram had only three levels: the root, the leaves, and one level in between. For tasks with more steps, the tree may be much taller.

A file written by human beings is called *source code*. Not all **`.c`** and **`.h`** files are source code: some are written by programs. For example, human beings write **`.y`** files and **`.l`** files and feed them to **`yacc`** and **`lex`**:

```
1$ yacc hoc.y          create y.tab.c: pp. 233−287
2$ lex lexer.l         create lex.yy.c: pp. 256−258
```

The resulting files **`y.tab.c`** and **`lex.yy.c`** must then be compiled into **`.o`** files. Here is a **`makefile`** for a tree with four levels:

```
OBJS = main.o y.tab.o lex.yy.o

prog: $(OBJS)
    $(CC) -o $@ $(OBJS) $(LOADLIBES)

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

y.tab.o: y.tab.c
    $(CC) $(CFLAGS) -c $*.c

lex.yy.o: lex.yy.c
    $(CC) $(CFLAGS) -c $*.c

y.tab.c: hoc.y
    $(YACC) $(YFLAGS) hoc.y

lex.yy.c: lexer.l
    $(LEX) $(LFLAGS) lexer.l
```

If the source files **hoc.y** and **lexer.l** were put under the protection of RCS, then **make** would need an additional preliminary step to get these files from RCS. The tree would then have five levels:

```
hoc.y:
    co hoc.y

lex.l:
    co lex.l
```

A forest

A *forest* is two or more trees. A **makefile** may contain a forest instead of a single tree. The root of one tree will usually be the executable file we want to create. The roots of the other trees may be also be files that we want to create, but more often are merely names for groups of commands that we want to execute.

For example, there is no file named **cleanup**, nor will there ever be. Type **make cleanup** to lead **make** to believe that we want to create a file named **cleanup**. **make** will then execute the indented **rm** command, believing that this will create **cleanup**.

```
OBJS = main.o file1.o
SOURCES = prog.h main.c file1.c

prog: $(OBJS)                    #The root of the first tree
    $(CC) $(CFLAGS) -o $@ $(OBJS)

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

file1.o: file1.c
    $(CC) $(CFLAGS) -c $*.c

#Remove all files that are not source code.
cleanup:
    rm prog $OBJS

print:
    pr -l60 $(SOURCES) | lpr     #minus lowercase L sixty

test:
    prog < test.data > test.out
    if cmp -s correct.out test.out;\
    then\
        rm test.out;\
        strip prog;\
        mv prog /usr/local/bin;\
    else\
        echo ’Failed the test.’;\
    fi
```

```
    1$ make
    2$ make main.o
    3$ make test
    4$ make cleanup
    5$ make print
```

Print only the files of which you have no up-to-date printout: p. 265

Add the following rule to the end of the first `makefile` in this handout.

```
print: func.h var.h main.c func.c var.c
    pr -l60 $? | lpr
    touch print
```

The first time you say `make print`, the indented commands will be executed because the file `print` does not exist. The macro `$?` will hold the names of all the dependents of this rule. The `touch` command will then create `print`.

Every subsequent time you say `make print`, the macro `$?` will hold the names of only those dependents that are newer than the file `print`. The indented command will print only the files that have been edited since the last time you said `make print`.

Use the same technique to back up only the files that have been modified since the last time they were backed up.
□