Summer 2013 Handout 6

Kill a process

A running program is called a *process*. We usually let a process live out its life and die a natural death. To kill it prematurely, type **`control-c`**; see Handout 1, p. 20, ¶ (3). But some programs (e.g., **`vi`**) are immune to this keystroke, and in any case the keystroke affects only a process running on your current terminal. How can we kill a process that is immune to **`control-c`** or running on another terminal?

Our example of a process on another terminal will be a shell. Suppose **`who`** says that you're logged into two different terminals, **`/dev/pts/62`** and **`/dev/pts/64`**. Apparently, you never logged out correctly the last time you logged in.

```
1$ who | sort | more
abc1234    pts/62       May 28 10:30      (DIALN-ASYNC600.DIAL.NET.NYU.EDU)
abc1234    pts/64       May 28 11:00      (HCGPC.EDLAB.ITS.NYU.EDU)
dube       pts/51       May 28 12:01      (DATA.ACF.NYU.EDU)
```

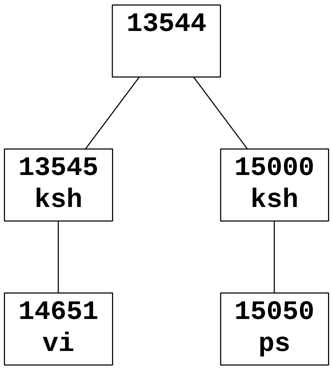
To log out from the terminal you are not using, kill the shell running there. Here are two ways to discover which terminal you *are* using:

```
2$ tty                                    Handout 5, pp. 17−18
/dev/pts/62

3$ who am i                               or who is god or who -m
abc1234    pts/62       May 28 10:30      (DIALN-ASYNC600.DIAL.NET.NYU.EDU)
```

They agree that I'm currently using terminal **`/dev/pts/62`**. Therefore **`/dev/pts/64`** is the one I want to log out of.

```
4$ ps -f | more           too few: only your programs running on your current terminal
5$ ps -Af | more                       too many: everyone's programs everywhere
6$ ps -Af | awk 'NR == 1 || $1 == "abc1234"' | more                    just right
     UID   PID  PPID  C    STIME TTY      TIME CMD
 abc1234 13545 13544  0 11:46:40 pts/64   0:00 -ksh
 abc1234 14651 13545  0 12:22:59 pts/64   0:00 vi handout6.ms
 abc1234 15000 13544  0 12:22:59 pts/62   0:00 -ksh
 abc1234 15050 15000  0 12:34:39 pts/62   0:00 ps -Af
```



To log out of terminal **`/dev/pts/64`**, **`kill`** the shell that is running there. This will also **`kill`** all of that shell's descendants, e.g., the **`vi handout6.ms`**. Minus Nine is the strongest kind of signal (“terminate with extreme prejudice”), pp. 150, 152. For the complete list of signal numbers, see **`signal`**(3head), the file **`/usr/include/sys/iso/signal_iso.h`**, or the shell command **`kill -l`** (minus lowercase L) in **`ksh`**(1) pp. 43−44. Handout 2, p. 3 lists the sections of the manual.

```
7$ kill -9 13545                    the PID number of the ksh on /dev/pts/64
8$ ps -Af | awk 'NR == 1 || $1 == "abc1234"' | more   make sure it was killed
9$ who | sort | more
```

After killing the shell and its descendants, you can recover the partially edited **`handout6.ms`** by

```
10$ mail                            A letter will tell you the name of the temp file.
11$ cd  back to the directory in which you were editing
12$ vi -r handout6.ms               the “recover” option
```

Run **`http://i5.nyu.edu/˜mm64/INFO1-CE9545/src/zombie.c`** to see a process that has turned itself into a zombie.

The dæmonic (DÆMONIC) ancestry of a process

Read from the bottom up to see the ancestry of a Korn Shell process. It may be different on other versions of Unix. **`STIME`** is starting time; **`TIME`** is cumulative elapsed time. **`ps`**(1) says the **`C`** field is obsolete. **`sched`** is the kernel; **`/etc/init`** is its only child. **`sshd`** is the Secure Shell dæmon. The dash in front of **`ksh`** means that this **`ksh`** is the shell that was launched when you logged in, not one that is merely running a shellscript.

```
1$ ps -Af | more                          output doctored by hand
     UID   PID  PPID   C    STIME TTY         TIME CMD
    root  1655  1655   0   Mar 29 ?           0:00 zsched
    root  2761  1655   0   Mar 29 ?           1:14 /usr/lib/ssh/sshd
    root 15976  2761   0 15:11:11 ?           0:00 /usr/lib/ssh/sshd
    mm64 15977 15976   0 15:11:11 ?           0:00 /usr/lib/ssh/sshd
    mm64 15978 15977   0 15:11:15 pts/9       0:00 -ksh
```

A process's PID is usually greater than its PPID, but it doesn't have to be.

```
2$ ps -Af | awk 'NR == 1 || $2 <= $3'
     UID   PID  PPID   C    STIME TTY         TIME CMD
    root  1655  1655   0   Mar 29 ?           0:00 zsched
    root  2480  2761   0   Apr 07 ?           0:00 /usr/lib/ssh/sshd
```

▼ Homework 6.1: print out the ancestry of your shell

Direct the output of **`ps -Af`** into a file. Use **`vi`** to remove all of the processes except your shell and all of its ancestors, and to reorder these processes in order of ancestry, as in the above example.

```
1$ cd
2$ pwd

3$ ps -Af > temp
4$ vi temp                                Doctor the output by hand.

5$ lpr temp
6$ rm temp                                or use the escape-. in Handout 2, p. 12
```

▲

Trap a signal

`Control-c` (signal number 2) usually kills a program immediately (Handout 1, p. 20, ¶ (3)). But if the following program is killed with a **`control-c`**, it will clean up after itself before it dies. See p. 151.

http://i5.nyu.edu/~mm64/INFO1-CE9545/src/lots

```
#!/bin/ksh
#Do lots of work.  Clean up if interrupted by a signal.

trap '
    echo
    echo I received control-c, which is signal number 2.

    if [[ -f ~/junk ]]
    then
        echo removing ~/junk
        rm ~/junk
    fi
    exit 2
' 2

date > ~/junk
echo Doing lots of work.
sleep 10
echo All done.
rm ~/junk
exit 0
```

```
1$ lots
Doing lots of work.
^C                                        you type a control-c
I received control-c, which is signal number 2.
removing /home1/a/abc1234/junk
```

!! commands in vi

Press **`ESC`** before typing the colon to make sure you're not in insert mode. Then move the cursor to an empty or superfluous line and type two exclamation points, a command, and **`RETURN`**. Only the second exclamation point will appear on the screen—the first one moves the cursor to the bottom line of the screen. The output of the command will replace the line where the cursor was.

1	`!!date`	
2	`!!cal 5 2013`	
3	`!!fonts`	*Insert output of your* **`fonts`** *shellscript into web page: Handout 4, pp. 11−12.*
4	`!!cat clipboard`	*Insert a copy of another file into the one you're editing.*
5	`!!cat -n clipboard`	*Insert a copy, with line numbers.*
6	`!!head clipboard`	*Insert only the first 10 lines of the file.*
7	`!!tail clipboard`	
8	`!!tail ~/clipboard`	*Looks for* **`clipboard`** *in your home directory, not your current directory.*

```
 9 !!grep word clipboard
10 !!grep word clipboard | head
11 !!grep word clipboard | head | cat -n
```

"Write" commands in vi, ed, and sed

Press **`ESC`** to make sure you're not in insert mode. There must be one blank before the filename. See the **`ed`** examples on pp. 12−13, 327−328, and the **`sed`** example on p. 112. To find the line number where the cursor is, use the **`control-g`** in Handout 3, p. 2.

1	`:20,30w clipboard`	*Create a file named* **`clipboard`** *containing a copy of lines 20−30.*
2	`:20,30w! clipboard`	*Overwrite the file if it already exists & you have permission to do so.*
3	`:20,30w >> clipboard`	*Append extra lines to an existing file.*
4	`:20,30w ~/clipboard`	*The file can be in any directory.*
5	`:20w clipboard`	*Copy only one line into the file.*
6	`:1,$w clipboard`	*Copy every line into the file.*
7	`:1,$-1w clipboard`	*Copy every line except the last one.*
8	`:1,$-2w clipboard`	*Copy every line except the last two.*
9	`:1,.w clipboard`	*Copy every line from the first line down to where the cursor is.*
10	`:.,$w clipboard`	*Copy every line from where the cursor is down to the last line.*
11	`:.,.+9w clipboard`	*Copy 10 lines, starting at the line where the cursor is.*
12	`:.-9,.w clipboard`	*Copy 10 lines, ending at the line where the cursor is.*

See **`/`** and **`?`** in Handout 3, p. 2:

13	`:.,/moe/w clipboard`	*Copy every line from the cursor down to the next* **`moe`**.
14	`:?moe?,.w clipboard`	*Copy every line from the previous* **`moe`** *to the cursor.*

Write any regular expression within the **`//`** or **`??`**. The following three lines are not complete **`vi`** commands.

15	`/^moe/`	*the next line that starts with* **`moe`**
16	`/moe$/`	*the next line that ends with* **`moe`**
17	`/^moe$/`	*the next line that consists entirely of* **`moe`**
18	`:?^{?,/^}/w clipboard`	*Copy the C function where the cursor is.*
19	`:?^{?-1,/^}/w clipboard`	*Copy the C function, including the line before the* **`{`**.

The lines you copy do not have to be consecutive:

20	`:g/moe/.w! >> clipboard`	*Copy every line that contains* **`moe`**.
21	`:g/moe/.+1w! >> clipboard`	*Copy every line that is one line after one that contains* **`moe`**.
22	`:g/moe/.+2w! >> clipboard`	*Copy every line that is two lines after one that contains* **`moe`**.
23	`:g/moe/.-1w! >> clipboard`	*Copy every line that is one line before one that contains* **`moe`**.
24	`:g/^{/.-1w! >> clipboard`	*Copy the name of every function in a C program.*

25 **`:v/moe/.w! >> clipboard`**	*Copy every line that doesn't contain* **`moe`**.

Instead of **`w clipboard`**, you can also print at the bottom of the screen. This has no effect on the file being edited. Then press **`RETURN`** again to make the list go away:

26 **`:g/moe/p`**	*Print every line that contains* **`moe`**.
27 **`:g/re/p`**	*where the word* **`grep`** *comes from: "global regular expression print"*
28 **`:g/^{/-1p`**	*Print the name of every function in a C program.*

The prefixes in front of the **`w`** and **`p`** commands can be used in front of any of the **`vi`** commands that start with a colon. Examples are the **`s`** command in **`vi`** in Handout 3, p. 4, and the **`y`** command in **`sed`** in Handout 4, p. 14.

Edit a C or C++ program

The leading number and blank at the start of each line are not part of the C program: they're printed only for your convenience.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 void f(void);
 5 void g(void);
 6
 7 int main(int argc, char **argv)
 8 {
 9     blah blah blah;
10     blah blah blah;
11     blah blah blah;
12     return EXIT_SUCCESS;
13 }
14
15 void f(void)
16 {
17     blah blah blah;
18     blah blah blah; {
19     blah blah blah; }
20 }
21
22 void g(void)
23 {
24     blah blah blah;
25     blah blah blah;
26     blah blah blah;
27 }
```

Stop and start programs with the Korn shell

See **`ksh`**(1) pp. 25−26, 39, 42, 43.

`1$ cd ~mm64/45/handout`	
`2$ vi handout6.ms`	
`control-z`	*Signal number 20;* **`ksh`***(1), p. 26; Handout 1, p. 20, ¶ (1)*
`Stopped`	*It says* **`Stopped`** *and the prompt reappears.*

```
3$ cd ˜mm64/45/lecture
4$ vi lecture6.ms               Start a second command running.
control-z                       Stop the second command, too.
Stopped                         It says Stopped with an uppercase S and the shell prompt reappears.

5$ jobs                    List the stopped jobs: Handout 2, p. 14, ll. 64−65.
[1]  - Stopped      vi handout6.ms          The [1] is not a PID number.
[2]  + Stopped      vi lecture6.ms

6$ fg %1                        to start job [1] running again (“foreground”)
7$ fg                           to start the most recently stopped job running again
```

The most recently stopped command is marked with a plus; the next-to-most-recent with a minus. The command **fg** (for *foreground*) will re-start the most recently stopped command. This is the **fg** in “What can go wrong” in Handout 1, p. 20, ¶ (1).

Run commands in the background: pp. 33−34

You can run two commands simultaneously by running the first one *in the background* with the ampersand in **ksh**(1) p. 1. The prompt will reappear immediately, allowing you to type the second command while the first is still running. A background command should always send its output to a file, not onto the screen.

```
1$ prog1 > ˜/prog1.out &
[1] 12345             It shows you the process ID number of the background process.
2$ prog2

3$ ps -f              See if prog1 is still running.
```

```
#!/bin/ksh
#What would go wrong without the quotes?

echo Please type 1 or 2 ’&’
echo press RETURN.
read n

exit 0
```

bg re-starts the most recently stopped program, this time in the background. This gives you a way to retroactively apply an ampersand to a previous command:

```
4$ prog > ˜/prog.out
control-z
Stopped
5$ bg                 You could also say fg.
6$                    The prompt reappears immediately because you said bg.
```

Outline of part 1 of this course

I. The file system

  A. Directory commands: **cd**, **pwd**, **mkdir**, **rmdir**.

  B. Special directories:

    1. **/** the root directory

    2. **.** your current directory

        3. `..` the parent of your current directory
        4. `~` your home directory
        5. `~abc1234` that person's home directory
    C. The arguments and output of **`ls -l`**; **`chmod`** the nine permission bits.
    D. Copy, move, rename, remove files; hard and symbolic links.
    E. Device "files" in **`/dev`**.
    F. Text files:
        1. **`/etc/passwd`** and **`/etc/group`**
        2. Message of the day: **`/etc/motd`**
        3. **`/usr/dict/words`** and **`/usr/dict/websters`**
        4. Web server: **`/var/apache2/2.2/logs/access_log`**, **`/etc/apache2/2.2/httpd.conf`**, **`~/public_html/index.html`**

II. Run a program
    A. Redirect the standard input, output and error output: **`|`**, **`>`**, **`<`**, **`>>`**, **`<<`**, **`2>`**, **`1>&2`**, **`‘`**back quotes**`‘`**.
    B. Produce and detect exit status: **`$?`**, **`exit 0`**, **`exit 1`**, **`if`**, **`set -e`**.
    C. Job control: **`control-c`**, **`control-z`**, **`fg`**, **`bg`**, **`&`**.

III. Unix utilities (tools)
    A. Programs that do output but no input: **`date`**, **`cal`**, **`who`**, **`ls -l`**, etc.
    B. Programs that do input and output (filters): **`sort`**, **`tr`**, **`grep`**, **`awk`**, **`wc -l`**, etc.

IV. Setting up a shellscript
    A. **`$PATH`**, **`which`**, **`whereis`**, **`~/bin`**, **`#!/bin/ksh`**, **`chmod`**, **`set -x`**.
    B. Command line arguments of a shellscript: **`$1`**, **`$2`**, **`$3`**, **`$*`**, **`$0`**, **`$#`**.
    C. World Wide Web gateways (CGI): **`Content-type`**, **`~/public_html/cgi-bin`**.
    D. Programs in other languages: C, C++, Perl, PHP, Python, Ruby, Java.

V. The shell language
    A. The **`.profile`** file.
    B. Local and environment variables: **`echo $x`**, **`export X=hello`**, **`env`**, **`let`**, **`read`**.
    C. Control structure: **`for`**, **`while`**, **`do`**, **`done`**, **`if`**, **`then`**, **`else`**, **`elif`**, **`fi`**, **`trap`**.

VI. **`vi`**
    A. **`vim -g -geometry=80x36`** (GUI)

Some Unix utilities that use regular expressions

`grep, egrep`	*search for a regular expression*
`more, less, pg, man, view`	*display text on the screen*
`ed, ex, vi, emacs`	*text editors*
`sed, awk`	*classic programmable filters: p. 101*
`Perl, Ruby, PHP`	*newer programming languages*
`dbx, lex, expr, csplit`	*miscellaneous*
`archie, nn`	*Internet*

In Java, import the **`java.util.regex`** package.

grep

```
1$ grep word filename
2$ grep word filename1 filename2 filename3

3$ cd /usr/share/groff/1.19.2/font/devps
4$ pwd

5$ grep internalname * | more
AB:internalname AvantGarde-Demi
ABI:internalname AvantGarde-DemiOblique
AI:internalname AvantGarde-BookOblique

6$ grep -h internalname * | more
internalname AvantGarde-Demi
internalname AvantGarde-DemiOblique
internalname AvantGarde-BookOblique
```

ˆ means "start of the line": p. 102

Search for your favorite prefixes: macro, mega, micro, octo, over, under, Italo, para, pre, pseudo, quasi, turbo, voodoo, etc.:

```
1$ grep -i 'ˆanti' /usr/dict/words
2$ grep -i 'ˆanti' /usr/dict/websters | more
3$ grep -i 'ˆanti' /usr/dict/words /usr/dict/websters | more
4$ grep -i 'ˆanti' /usr/dict/w*s | more          easier way to do the same thing
5$ awk '/ˆanti/' /usr/dict/w*s | more                        case sensitive
6$ perl -ne 'print if /ˆanti/i;' /usr/dict/w*s | more      case insensitive

7$ ls -l | tail +2 | grep 'ˆ-'    Output only the names of files: Handout 1, p. 10; 4:17; 5:16
8$ ls -l | tail +2 | grep 'ˆd'    Output only the names of subdirectories.

9$ grep 'ˆ#' prog.c | lpr         Print the lines that start with #
10$ grep -v 'ˆ#' prog.c | lpr     Print lines that don't start with #; Handout 5, p. 15 for -v
```

$ means "end of the line": p. 102

Search for you favorite suffixes: able, mania, maniac, ocracy, oid, phobiac, oxide, tomy (as in "lobotomy"), ist, ity, esque, fish, etc.:

```
1$ grep 'phobia$' /usr/dict/words
2$ grep -i 'phobia$' /usr/dict/w*s | more

3$ grep 'ˆ$' file.txt | wc -l       How many empty lines does file.txt contain?
4$ grep -v 'ˆ$' file.txt | wc -l    How many non-empty lines?

5$ grep '\$100'                   search for a dollar sign and a hundred
6$ grep '\ˆ'                      search for a caret
7$ grep '\\'                      search for a backslash
8$ grep '\\\\'                    search for two consecutive backslashes
```

```
9$ grep '^\^'                  lines that begin with a caret
10$ grep '^\\\^'               lines that begin with a backslash and a caret
11$ grep '\$$'                 lines that end with a dollar sign
12$ grep '\\\$$'               lines that end with a backslash and a dollar sign
```

The . wildcard: p. 103

Example 3 would output all the lines output by examples 1 or 2, but it wouldn't output **`septirate`** or **`seprate`**.

```
1$ grep 'seperate' /usr/dict/words
2$ grep 'separate' /usr/dict/words
3$ grep 'sep.rate' /usr/dict/words
4$ grep '\.' /usr/dict/words          search for a dot

5$ grep -hi '^...u.$' /usr/dict/w*s | more        Why do I need the anchors?
6$ grep -hi  '...u.'  /usr/dict/w*s | more

7$ grep -i '^b.g$' /usr/dict/words
8$ grep -i '^p.t$' /usr/dict/words
```

In Handout 6, p. 8, second line 5, the backslash turned off the special meaning that a character (**`$`**) had to **`grep`**. In the following line 11, the backslashes turns *on* the special meaning that the characters **`{`** and **`}`** have to **`grep`**. The curly braces were added to **`grep`** as an afterthought.

```
9$ grep '^.....$'              Output all lines consisting of exactly five characters.
10$ grep -v '^.....$'          Output all lines consisting of more or less than five characters.
11$ grep -v '^.\{5\}$'         regexp(5) pp. 2−3, ¶ 2.3

12$ grep '.....'               Output all lines consisting of 5 or more characters.
13$ grep -v '.....'            Output all lines consisting of less than 5 characters.

14$ grep '......'              Output all lines consisting of more than 5 characters.
15$ grep -v '......'           Output all lines consisting of 5 or less characters.
```

We listed the terminals and wrote to them in Handout 2, pp. 19−20. A terminal not in use is owned by **`root`**. List the terminals in use that anyone can write to (i.e., whose third **`w`** bit is on):

```
16$ ls -l /dev/pts | awk 'NR >= 2 && $3 != "root"' |
    grep '^c.\{7\}w' | more
```

The **`-d`** option of **`ls`** was in Handout 1, p. 10:

```
17$ awk -F: '{print $6}' /etc/passwd | head -3
/root
/
/usr/bin
```

```
18$ ls -ld `awk -F: '{print $6}' /etc/passwd` | grep '^d.\{7\}w' | more
drwxrwxrwx   4 yb610    users         35 Aug  5  2012 /home1/y/yb610
```

```
19$ ls -ld `awk -F: '{print $6}' /etc/passwd` | grep '^d.\{7\}w' |
awk '{print $3}' > ~/names

20$ mail `cat ~/names` < ~/letter
21$ rm ~/names
```

Avoid creating **`~/names`** by nesting the back quotes. The **`$( )`** notation in Handout 5, p. 15 is easier to nest:

```
22$ mail $(ls -ld $(awk -F: '{print $6}' /etc/passwd) |
    grep '^d.\{7\}w' | awk '{print $3}') < ~/letter
```

▼ Homework 6.2: Capitalism, Communism, Nationalism

The Twentieth Century was a battleground of conflicting isms, from absenteeism to Zionism. List the 97 isms (words ending with "ism", either upper or lowercase) in **`/usr/dict/words`** that are at least six characters long. You get no credit if you use more than one **`grep`**, or if you use **`tr`**, or if your regular expression contains an asterisk, or if you print the wrong number of isms, or if you don't hand in a printout of the output, or if your output contains the word "Page". You get no credit if you printed "prism" or "numismatic", or if you failed to print "sadism" and "fascism". Pipe the output of **`grep`** into a **`pr`** whose second option is minus lowercase i quote blank quote one, third is minus lowercase L 25 (Handout 4, pp. 2−3), and fourth is minus lowercase W eighty.

```
pr -4 -i' '1 -l25 -t -w80
```

to output the isms in four columns of 25 lines each. The output must be printed in a monospace font to make the columns line up. You get no credit if you do not give **`pr`** the five arguments shown above.

▲

The [] wildcard: pp. 102−103

`.`	*any character at all*
`[ABCDEFGHIJKLMNOPQRSTUVWXYZ]`	*a more selective wildcard: any uppercase letter*
`[A-Z]`	*any uppercase letter; no space on either side of dash, can't say* **`[Z-A]`**

```
#!/bin/ksh
#Output the last name of each student, one per line.
#It's the last word on the second line of each bio file.
#Why do we {print $NF} (p. 115) instead of {print $2} or {print $3}?

for filename in ~mm64/public_html/INFO1-CE9545/bio/*
do
    awk 'NR == 2 {print $NF}' $filename
done
```

```
Berezovskiy
chennankara
sarika
Panopoulos
Poltiyelov

1$ grep -i '^[A-L]' ~/names > ~/al
2$ grep -i '^[M-Z]' ~/names > ~/mz

3$ grep '[a-z]'             any lowercase letter; can't say [z-a]

4$ grep '[0123456789]'      any decimal digit (10 possibilities)
5$ grep '[0-9]'             any decimal digit (10 possibilities)
6$ grep '19[0-9][0-9]'      prepare for the millennium; not necessarily two copies of the same digit
7$ grep -hi '[ct][sz]ar' /usr/dict/w*s | sort | uniq   in search of Russia's imperial past
```

```
8$ grep '[0-7]'              any octal digit (8 possibilities)
9$ grep '[0-9A-Fa-f]'        any hexadecimal digit (22 possibilities: 10 + 6 + 6); can also use -i
10$ grep '[A-Za-z0-9_]'      any char allowed in shell variable name (63 possibilities: 26 + 26 + 10 + 1)

11$ grep '[02468]$'          search for an even number (assume each line ends with a number)
12$ grep '^[0-9][0-9][0-9][0-9][0-9]$'              United States zip code: 10040
13$ grep '^[0-9]\{5\}$'
14$ grep '^[A-Z][0-9][A-Z] [0-9][A-Z][0-9]$'        Canadian postal code: A2B 3C4

15$ grep '^[A-CEGHJ-NPR-TV-Z][0-9][A-CEGHJ-NPR-TV-Z] [0-9][A-CEGHJ-NPR-TV-Z][0-9]$'
```

Line 16 is a simpler way to do line 15. But to output one copy of each line that is not output by 15, we can add a **-v** to 15. There is no easy way to output one copy of each line that is not output by 16.

```
16$ grep '^[A-Z][0-9][A-Z] [0-9][A-Z][0-9]$' | grep -v '[DFIOQU]'     6 forbidden letters

17$ grep '\['                search for a [
```

The **awk** outputs the nine characters starting with the second character on each line of input:

```
18$ ls -l | awk 'NR >= 2 {print substr($1, 2, 9)}' | more
rw-r--r--        okay
rwx------        okay
---r--r--        bad: heavier on the right
r--r-xrw-        bad: heavier on the right
```

The **--** argument means that none of the following arguments are options, even if they begin with a minus.

```
19$ ls -l | awk 'NR >= 2 {print substr($1, 2, 9)}' | grep -- '-..[rwx]' | more
20$ ls -l | tail +2 | grep '^.\{1,6\}-..[rwx]' | more
```

A trick question

Would a line consisting of only the three characters **ABC** be output if you fed it into the following **grep**?

```
echo ABC | grep '^[ABC]$'
```

What if you removed the square brackets? What if you restored the square brackets but omitted one or both of the anchors **^** and **$**? What if you removed the square brackets and the anchors?

▼ Homework 6.3: social security numbers

The file **$S45/socialsecurity** should contain one social security number per line: just nine digits, with no blanks, dashes, or anything else. Write a shellscript which outputs the 7 out of the 9 lines in this file that do not match this pattern. Use **grep -v**.
▲

▼ Homework 6.4: Bonfire of the Vanities (Tom Wolfe)

> Before losing consciousness, he was able to give his mother the first letter—R—and five possibilities for the second letter—E, F, B, R, P—of the license plate of the luxurious Mercedes-Benz that ran him down on Bruckner Boulevard and sped off.
>
> *—Chapter 12*

Write a shellscript that will output the number of words in **/usr/dict/words** and **/usr/dict/websters** that match the above description and that are at most seven characters long. Output only one line, containing only one number. Do not output the words themselves.

Use exactly one **`grep`** and no **`awk`**. Give it **`-h`** and **`-i`** options, combined to **`-hi`**. You get no credit unless you end up with 1270 such words after you **`sort +0f +0`** (p. 106) and **`uniq`** them. Search for both upper and lowercase: don't miss **`RPM`**. Do not impose a minimum length of three: don't miss **`re`**.

▲

▼ Homework 6.5: 1−800−737−3783

Write a shellscript named **`spelledby`** to output all the words in **`/usr/dict/words`** and **`/usr/dict/websters`** that are spelled by your seven-digit phone number. Use someone else's number if yours contains a zero or one. This shellscript should take no input and no command line arguments. Search for both upper and lowercase. Use **`sort +0f +0`** (p. 106) and **`uniq`** to output only one copy of each word, in alphabetical order. Do not output the name of any dictionary.

No luck? Try your first three and last four digits separately; or your first four and last three, etc. Remove the **`^`** and **`$`** only as a last resort.

2	abc	6	mno
3	def	7	pqrs
4	ghi	8	tuv
5	jkl	9	wxyz

▲

▼ Homework 6.6: administrative aid for the NYU School of Continuing and Professional Studies

Write a shellscript named **`daycount`** that will output how many of the specified week days there are in a given month. The first argument must consist of exactly one uppercase letter followed by exactly two lowercase letters. Output only one line, containing only one number. For example,

```
1$ daycount Tuesday 5 2013
4
```

This shellscript must take exactly three command line arguments: output an error message otherwise. Do not write separate error messages for "too many" and "too few". An error message must have the three trimmings on the error message in the shellscript in Handout 5, p. 5; you get no credit otherwise.

(1) The error message must begin with the name of the program (**`$0`**), a colon, and a blank.

(2) The error message must be directed to the standard error output (**`1>&2`**) not the standard output.

(3) The program must yield a non-zero exit status in case of error, 0 otherwise.

Do not use a **`read`** statement or a **`set`** statement. Do not forget Saturday and Sunday. Do not use **`awk`**. If the user typed the three command line arguments correctly, **`daycount`** must output only one line, containing only one number and nothing else.

To count how many Sundays there are, count the lines that have a digit as their second character:

```
2$ cal 5 2013
   May 2013
 S  M Tu  W Th  F  S
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31

3$ cal 5 2013 | tail +3 | grep '^.[0-9]' | wc -l
4
```

To count the Mondays, search for lines that have a digit as the fifth character. (This number may be different on other machines.) Write a chain of **`if-then-elif`** statements as in Homework 5.8 (Handout 5, pp. 22−23) to compare the first argument with each day:

```
    if [[ $1 == Sunday ]]
    then
```

Format and indent your chain of **`if-then-elif-else-fi`** statements in exactly the same way as the last box in Handout 4, p. 20; you get no credit if there is any difference at all.

Add a final **`else`** containing another error message in case the user types a bad first argument, which you must output as part of the error message. You get credit only if the error message includes the bad first argument. This error message must also have the three trimmings listed above. Use six **`elif`**'s instead of six **`else if`**'s—you get no credit if your shellscript has more than two **`fi`**'s.

Which way to find the Saturdays is the easiest to proofread? **`grep`** has **`\{ \}`** but no parentheses, **`egrep`** has parentheses but no **`\{ \}`**. Perl has both, and you don't even need the backslashes.

```
    4$ grep '^...................[0-9]'
    5$ grep '^.. .. .. .. .. .. .[0-9]'
    6$ grep '^.\{19\}[0-9]'
    7$ perl -ne 'print if /^(.. ){6}.\d/;'
```

✎ For a more compact notation, use **`case`** instead of **`if-then-elif`**. See pp. 134−135, **ksh**(1) p. 2.
▲

[^] wildcard: pp. 102−103

The whitespace in line 2 is one blank and one tab; the **`\'`** stands for one single quote. The **`\'`** is surrounded by a pair of single quotes, and is therefore outside of the single-quoted territory.

```
1$ grep '[A-Z]'              any uppercase letter
2$ grep '[]  !"#$%&'\''()*+,./0123456789:;<=>?@[\^_`abcdefghijklmnopqrstuvwxyz{|}~-]'
3$ grep '[^ABCDEFGHIJKLMNOPQRSTUVWXYZ]'         any character except an uppercase letter
4$ grep '[^A-Z]'             any character except an uppercase letter

    5$ grep '[^acgt]'        any character except a lowercase a, c, g, or t
    6$ grep '[^A]'           any character except an uppercase A

    7$ grep -hi 'q[^u]' /usr/dict/w*s | sort | uniq | more
    QED
    Qatar

    8$ grep -hi 'q$' /usr/dict/w*s | sort | uniq | more
    IQ
    Iraq

    9$ awk -F: '{print $1}' /etc/passwd | grep '[^a-z0-9]'
```

The difference between [^ and -v

```
    1$ echo ABC | grep '[^A]'       every line that contains a character other than A
    2$ echo ABC | grep -v 'A'       every line that has no A; quotes unnecessary
```

Search for a number or word that is not part of a longer number or word

To search for the number **`100`** without finding longer numbers such as **`2100`** or **`21003`**,

```
1$ grep        100
2$ grep '[^0-9]100[^0-9]'             Does this output every line that contains 100?
3$ grep      '^100[^0-9]'
4$ grep '[^0-9]100$'
5$ grep      '^100$'
```

To search for a variable named **max** without finding longer names such as **maximum** or **xmax**, see Handout 6, p. 10, line 11.

```
6$ grep              max
7$ grep '[^A-Za-z0-9_]max[^A-Za-z0-9_]'
8$ grep             '^max[^A-Za-z0-9_]'
9$ grep '[^A-Za-z0-9_]max$'
10$ grep            '^max$'

11$ grep -w max                        a word all by itself
12$ grep '\<max\>'                     in other versions of Unix
```

Search for a non-printable character

character	base 2 binary	base 8 octal	base 10 decimal	base 16 hexadecimal
'A'	**01000001**	**101**	**65**	**41**
BEL	**00000111**	**7**	**7**	**7**

The character **'A'** has ASCII code 65. The invisible character that rings the bell has ASCII code 7. See **ascii**(5) in Handout 2, p. 1. **echo** requires a backslash and a zero in front of an octal number:

```
1$ echo '\0101'
A

2$ echo '\07'                          Beep the terminal (or '\a').
```

Not counting the tab character, the printable characters with the smallest and largest ASCII codes are the blank and tilde. Their ASCII codes are 32 and 126 respectively. Unfortunately, the ASCII code of the tab is 9.

The wildcard

```
3$ grep '[ -~]'                        one blank after the [
```

would therefore match any printable character except for the tab, and the wildcard

```
4$ grep '[ -~	]'                       one blank after the [, one tab before the ]
```

would match any printable character. To search for lines containing a non-printable character,

```
5$ grep '[^ -~	]'                      one blank after the ^, one tab before the ]
```

▼ Homework 6.7: search for unusual execute permission

The fourth character of every line (except the first) output by **ls -l** is a dash or a lowercase **x**, right?

```
1$ cd ~mm64/45/handout
2$ ls -l | tail +2
-r--r--r--   1 mm64     users      59587 Aug 25  2012 handout6.ms
-r-xr--r--   1 mm64     users       1122 Aug 24  2004 indexscript
```

grep for all the lines in the output of **ls -l /usr/bin | tail +2** whose fourth character is neither a lowercase **x** nor a dash. On May 28, 2013, the fourth character of 14 of the 1078 lines of output was neither a lowercase **x** nor a dash. See set-uid on pp. 54−55.

Use only a single **grep**. Do not use the **-v** option of **grep**. You get no credit if your regular expression contains the letters **w** or **y**. You get no credit if you make the ☜ mistake in Handout 6, p. 14, the second line 3.

Verify that your **grep** outputs 94 of the 98 lines in the file **$S45/unusual.execute**. Don't hand it in until it does. Do not hand in any line from the file **$S45/unusual.execute**.
▲

What can go wrong with a wildcard with [square brackets]

Never use **[**square brackets**]** unless you are writing more than one character in them:

```
1$ grep '[ABC]'     a wildcard to match any one of the 3 characters A, B, or C
2$ grep '[AB]'      good
3$ grep '[A]'       bad      ☜
4$ grep 'A'         good
```

The dash has a special meaning only when it is within **[**square brackets**]**. You need no dash when the surrounding characters are consecutive:

```
5$ grep '[ABC]'     a wildcard to match any one of the 3 characters A, B, or C
6$ grep '[A-C]'     good
7$ grep '[A-B]'     bad
8$ grep '[AB]'      good
```

The three characters **-^]** are the only ones that have a special meaning within square brackets. Lines 11, 14, and 17 below show how to turn off their special meaning. See the top of p. 103 in the textbook and **regexp**(5) p. 2.

```
9$ grep '[A-C]'     match any one of the 3 characters A, B, or C.
10$ grep '[AC-]'    match any one of the 3 characters A, C, or -.
11$ grep '[-AC]'    match any one of the 3 characters -, A, or C.

12$ grep '[^BC]'    match any character except a B or C.
13$ grep '[B^C]'    match any one of the 3 characters B, ^, or C.
14$ grep '[BC^]'    match any one of the 3 characters B, C, or ^.

15$ grep '[AB]]'    match A or B, followed by ].
16$ grep '[]AB]'    match any one of the 3 characters ], A, or B; Handout 6, p. 12, first line 2
17$ grep '[^]]'     match any character except ].

18$ grep '[]^-]'    match any one of the 3 characters ], ^, or -
19$ grep '[][]'     match either ] or [
20$ grep '[^[]'     match any character except [
21$ grep '[^][]'    match any character except ] or [
```

Never use a **** within square brackets to turn off the special meaning of any regular expression character such as **.**, **^**, **$**, etc.:

```
22$ grep '[^.]trash'  Search for trash preceded by a character other than a dot.
23$ grep '[^$]'       lines that contain a character other than a dollar sign
24$ grep '[^^]'       lines that contain a character other than a caret
25$ grep '^[^^]'      lines that begin with a character other than a caret
26$ grep '^[^^]\^'    lines that begin with a character other than a caret, followed by a caret
```

Asterisk in a regular expression: p. 103

A blank and an asterisk mean “zero or more consecutive blanks”. Never write an asterisk in a regular expression without a character or wildcard immediately to the left of it. `.*` (i.e., dot star) in a regular expression has the same meaning as `*` in the shell language.

```
1$ grep -i    '^[A-L]' ~/names >  ~/al           no leading blanks
2$ grep -i   '^ [A-L]' ~/names >> ~/al           one leading blank
3$ grep -i  '^  [A-L]' ~/names >> ~/al           two leading blanks
4$ grep -i '^   [A-L]' ~/names >> ~/al           three leading blanks

5$ grep -i '^ *[A-L]' ~/names > ~/al             zero or more leading blanks

6$ grep '21210040'             separated by no characters
7$ grep '212.10040'            separated by one character
8$ grep '212..10040'           separated by two characters
9$ grep '212...10040'          separated by three characters

10$ grep '212.*10040'          separated by zero or more characters

11$ grep -i '^anti.*ism$' /usr/dict/websters
12$ grep -i '^anti' /usr/dict/websters | grep -i 'ism$'

13$ grep '\*'                  search for an asterisk
```

Assuming that any space in the sixth field is surrounded by two words,

```
14$ awk -F: '{print $5}' /etc/passwd | grep ' .* .* ' | more
SendMail Message Submission Program
ZFS Automatic Snapshots Reserved UID
UPnP Server Reserved UID
NFS Anonymous Access User
```

Regular expressions using [^] and *: pp. 102−103

To output the lines that contain phone numbers in area code 212 (i.e., lines that contain **212** with no digits ahead of them), see Handout 1, p. 8.

To output the lines in the file **/etc/passwd** with no password,

```
1$ grep '::' /etc/passwd
2$ grep '^[^:]*::' /etc/passwd
3$ awk -F: '$2 == ""' /etc/passwd       awk is easier: Handout 4, p. 18, lines 9−10
```

Fold long lines (Handout 5, p. 11, Homework 5.4, ¶ (3)) at a width of 90 bytes:

```
4$ tail -3 /var/apache2/2.2/logs/access_log | fold -b -90
100.2.35.247 - - [28/May/2013:15:13:53 -0400] "GET /~ajg494/Final/paperbackground.jpg HTTP
/1.1" 200 89233
128.122.108.95 - - [28/May/2013:15:17:38 -0400] "GET / HTTP/1.1" 200 6311
99.186.96.72 - - [28/May/2013:15:22:41 -0400] "GET /~ajg494/Final/paperbackground.jpg HTTP
/1.1" 200 89233
```

Does every line have exactly two double quotes, no more and no less? In other words, is every line in the format **—"—"—**, where each **—** represents a region that has no double quotes?

```
5$ grep -v ’^[^"]*"[^"]*"[^"]*$’ /var/apache2/2.2/logs/access_log |
    tail -3 | fold -b -90
6$ awk ’-F"’ ’NF != 3’ /var/apache2/2.2/logs/access_log      NF is the number of fields
49.212.155.117 - - [23/May/2013:19:14:44 -0400] "GET /~myd212/works.html\\\\\" HTTP/1.1
04 224
49.212.156.175 - - [24/May/2013:14:33:30 -0400] "GET /~myd212/works.html\\\\\" HTTP/1.1
04 224
```

Does every line have exactly one left square bracket followed by exactly one right square bracket, with no other square brackets? In other words, is every line in the format **—[—]—**, where each **—** represents a region that has no square brackets?

```
7$ grep -v ’^[^][]*\[[^][]*\][^][]*$’ /var/apache2/2.2/logs/access_log |
    tail -3 | fold -b -90
98.14.224.61 - - [06/May/2013:10:57:49 -0400] "GET /~sws272/a[i];}} HTTP/1.1" 404 213
98.14.224.61 - - [06/May/2013:10:57:50 -0400] "GET /~sws272/a[i+2];} HTTP/1.1" 404 214
```

One or more

See **regexp**(5), pp. 2−3, ¶ 2.3. Output the lines that have a negative number (a dash followed immediately by one or more consecutive digits) followed immediately by a blank.

```
1$ grep -- ’-[0-9][0-9]* ’
2$ grep -- ’-[0-9]\{1,\} ’            Handout 6, p. 9, line 11; Handout 6, p. 11, lines 22−23
3$ egrep -- ’-[0-9]+ ’
4$ grep -- ’-[1-9][0-9]* ’            disallow negative zero -0
```

▼ Homework 6.8: a facetious example

Write a shellscript to output all the words in **/usr/dict/words** and **/usr/dict/websters** that contain all five vowels in alphabetical order, with each vowel appearing exactly once (e.g., abstemious). "Y" is not a vowel. Search for both upper and lowercase.

These words are in the format **—a—e—i—o—u—**, where each **—** represents a region that has no vowels. **[**Square brackets**]** are unnecessary around a single character: use them only around a group of characters. You get no credit if you make the ☜ mistake in Handout 6, p. 14, the second line 3. Do not use **\(\) \1**. You must specify that the part of the word before the "**a**" consists only of zero or more consonants. And you must specify that the rest of the word after the "**u**" consists only of zero or more consonants. Output only those lines in **/usr/dict/words** and **/usr/dict/websters** that are occupied entirely by the characters you specified. Add whatever is necessary to ensure that these characters stretch from the beginning of the line to the end.

Split a long command line with **** *between* arguments, not in the middle of an argument. **grep** should output one line of **/usr/dict/words** and 16 lines of **/usr/dict/websters**. Verify that it outputs 8 of the 24 lines in the file **$S45/vowel**. Don't hand it in until it does.

▲

grep for the most common bug in C, C++, Java, and Perl

To output the lines in **prog.c** that contain **if (a = b)**, where **a** and **b** are any expressions,

```
1$ grep       ’if *(.*=.*)’     prog.c
2$ grep    ’if *(.*[^=]=[^=].*)’ prog.c
3$ grep ’if *(.*[^=!<>]=[^=].*)’ prog.c
```

Allow zero or more blanks between the word **if** and the left parentheses. The first **grep** finds lines such as

```
if (a = b)
```

but unfortunately it also finds correct lines such as

```
if (a == b)
```

—exactly what we want to avoid. The second **grep** is smart enough to avoid **if (a == b)**, but unfortunately it still finds correct lines such as

```
if (a != b)
if (a <= b)
if (a >= b)
```

(C connoisseurs will recognize that we must also avoid **if (a += b)**, etc.)

How not to use *

Here are three ways of doing the same thing. Write only the first one.

```
1$ grep 'moe'           good
2$ grep 'moe.*'         bad
3$ grep 'moe.*$'        bad
```

Similarly, here are three ways of doing the same thing. Write only the first one.

```
4$ grep    'moe'        good
5$ grep  '.*moe'        bad
6$ grep '^.*moe'        bad
```

Why aren't the following lines three ways of doing the same thing?

```
7$ echo 3212 | grep         '212'
8$ echo 3212 | grep  '[^0-9]*212'
9$ echo 3212 | grep '^[^0-9]*212'
```

Examples of touch-sensitive client-side imagemaps

http://i5.nyu.edu/~mm64/imagemap.html
http://i5.nyu.edu/~mm64/INFO1-CE9545/00120132.html*Unix Section 1 (Tuesday)*
http://i5.nyu.edu/~mm64/INFO1-CE9264/00120132.html*C++ Part I Section 1 (Monday)*
http://i5.nyu.edu/~mm64/INFO1-CE9236/00120132.html*iPhone and iPad App Development Section 1 (Thursday*

Official documentation for HTML 4.0 client-side touch-sensitive imagemaps

http://www.w3.org/TR/REC-html40/struct/objects.html#h-13.6

Create a client-side touch-sensitive imagemap

Let's assume you have a picture in a file named **~/public_html/picture.gif**. Write the following in your **~/public_html/index.html** file.

Within the pair of **MAP** tags you can have as many **CIRCLE**'s and **RECT**angle's as you like, but only at most one **DEFAULT**. Put the **DEFAULT** last; it's "none of the above" area.

Coördinates are measured in pixels and are always greater than or equal to zero. Put no space adjacent to a comma in the coördinates. The three coördinates of a **CIRCLE** are

(1) the horizontal distance from the left edge of the image to the center of the circle;

(2) the vertical distance from the top edge of the image to the center of the circle;

(3) the radius of the circle.

The four coördinates of a **RECT**angle are

(1) the horizontal distance from the left edge of the image to the left edge of the rectangle;

(2) the vertical distance from the top edge of the image to the top edge of the rectangle;

(3) the horizontal distance from the left edge of the image to the right edge of the rectangle;

(4) the vertical distance from the top edge of the image to the bottom edge of the rectangle.

The coördinates of a **POLY**gon are

(1) The coördinates of one vertex.

(2) The coördinates of the next vertex, etc.

The statements within the **"**double quotes**"** are written in the language JavaScript. The string assigned to **status** is displayed in the status bar at the bottom of the browser window. See

https://developer.mozilla.org/En/DOM:window.status

```
<!-- Excerpt from your ˜/public_html/index.html file. -->

<MAP NAME = "mymap">

<AREA
SHAPE = CIRCLE
COORDS = 253,146,10
HREF = "http://i5.nyu.edu/~abc1234/"
onMouseOver = "status = ’Visit my home page.’; return true;"
onMouseMove = "status = ’Visit my home page.’; return true;"
onMouseOut = "status = ’’;">

<AREA
SHAPE = RECT
COORDS = 10,10,20,30
HREF = "http://i5.nyu.edu/cgi-bin/cgiwrap/abc1234/mygateway"
onMouseOver = "status = ’Run my gateway.’; return true;"
onMouseMove = "status = ’Run my gateway.’; return true;"
onMouseOut = "status = ’’;">

<AREA
SHAPE = POLY
COORDS = 10,10,20,10,20,20
HREF = "http://i5.nyu.edu/cgi-bin/donothing"
onMouseOver = "status = ’Not a touch-sensitive area.’; return true;"
onMouseMove = "status = ’Not a touch-sensitive area.’; return true;"
onMouseOut = "status = ’’;">

<AREA
SHAPE = DEFAULT
HREF = "http://i5.nyu.edu/other.html"
onMouseOver = "status = ’Visit some other page.’; return true;"
onMouseMove = "status = ’Visit some other page.’; return true;"
onMouseOut = "status = ’’;">

</MAP>

<P>
Click on various points in this image:
<BR>
```

```
<IMG
SRC = "http://i5.nyu.edu/~abc1234/picture.gif"
ALT = "[Picture of my dog in Colorado]"
USEMAP = "#mymap">
```

How do you discover the coördinates?

Before you do any of the above, put the image in your home page with the following **`ISMAP`** instead of **`USEMAP = "#mymap"`**. And temporarily surround the **`IMG`** tag with the following pair of **`A`** tags. This makes it a ‘‘server-side’’ rather than a ‘‘client-side’’ imagemap:

```
<!-- Excerpt from your ~/public_html/index.html file. -->

<P>
Click on various points in this image:
<BR>
<A HREF = "http://i5.nyu.edu/cgi-bin/donothing">
<IMG
SRC = "http://i5.nyu.edu/~abc1234/picture.gif"
ALT = "[Picture of my dog in Colorado]"
ISMAP>
</A>
```

Then press the browser’s **`Reload`** button. Click on the image and write down the *x*, *y* coördinates that you see after the **`?`** at the end of the **`Location`** field at the top of the browser window.

When you’ve discovered the coördinates of all the points you need, remove the pair of **`A`** tags, change the **`ISMAP`** to **`USEMAP = "#mymap"`**, and write the list of **`AREA`**’s within the pair of **`MAP`** tags. Remember to **`Reload`** again.

Output a list of HTML areas

```
abc1234 100 200
def5678 300 400
ghi9012 500 600
```

Within a **`"`**double quoted**`"`** string in the argument of **`awk`**, there must be a backslash immediately before each double quote. A long **`print`** statement can be split into two lines with a backslash. There must be nothing to the right of the backslash, not even a blank. For the absence of a comma between the items given to **`print`**, see Handout 3, p. 26; Handout 4, p. 16; Handout 5, p. 12.

```
#!/bin/ksh

awk '{print "<AREA SHAPE = CIRCLE COORDS = " $2 "," $3 \
    ",10 HREF = \"/~" $1 "/\">"}'

exit 0
```

```
<AREA SHAPE = CIRCLE COORDS = 100,200,10 HREF = "/~abc1234/">
<AREA SHAPE = CIRCLE COORDS = 300,400,10 HREF = "/~def5678/">
<AREA SHAPE = CIRCLE COORDS = 500,600,10 HREF = "/~ghi9012/">
```

□