EE324 – Applications Programming for Embedded Systems

Lecture 02(a)

Introduction to Shell Programming

2.1 – I/O and Redirection

2.2 – Regular Expressions

2.3 – The Shell Enviornment

2.4 – Processes & Job Control

2.5 – Shell Programming I

2.6 – Shell Programming II

I/O and Redirection

Standard I/O

- Standard Output (stdout)
  - default place to which programs write
- Standard Input (stdin)
  - default place from which programs read
- Standard Error (stderr)
  - default place where errors are reported
- To demonstrate -- cat
  - Echoes everything you typed in with an <enter>
  - Quits when you press Ctrl-d at a new line -- (EOF)

Redirecting Standard Output

- cat file1 file2 > file3
  - file3 is created if not there
  - concatenates file1 and file2 into file3
- cat file1 file2 >! file3
  - file3 is clobbered if there
- cat file1 file2 >> file3
  - file3 is created if not there
  - file3 is appended to if it is there
- cat > file3
  - file3 is created from whatever user provides from standard input
  - Makes for a cheap text editor!

Redirecting Standard Error

- Generally direct standard output and standard error to the same place:

  ```
  obelix[1] > cat myfile >& yourfile
  ```

  - If myfile exists, it is copied into yourfile
  - If myfile does not exist, an error message

    ```
    cat: myfile: No such file or directory
    ```

    is copied in yourfile

- In tcsh, to write standard output and standard error into different files:

  ```
  obelix[2] > (cat myfile > yourfile) >& yourerrorfile
  ```

- In sh (for shell scripts), standard error is redirected differently … more on this later.

Redirecting Standard Input

- obelix[1] > cat < oldfile > newfile
- A more useful example:
  - obelix[2] > tr string1 string2
    - Read from standard input.
    - Character *n* of string1 translated to character *n* of string2.
    - Results written to standard output.
  - Example of use:

```
obelix[3] > tr aeoiu eoiua
obelix[4] > tr a-z A-Z < file1 > file2
```

/dev/null

- /dev/null
  - A virtual file that is always empty.
  - Copy things to here and they disappear.
    - cp myfile /dev/null
    - mv myfile /dev/null
  - Copy from here and get an empty file.
    - cp /dev/null myfile
  - Redirect error messages to this file
    - (ls -l > recordfile) >& /dev/null
    - Basically, all error messages are discarded.

Filters (1)

- Filters are programs that:
  - Read stdin.
  - Modify it.
  - Write the results to stdout.
- Filters typically do not need user input.
- Example:
  - tr (translate):
    - Read stdin
    - Echo to stdout, translating some specified characters
- Many filters can also take file names as operands for input, instead of using stdin.

Filters (2)

- grep patternstr:
  - Read stdin and write lines containing patternstr to stdout

```
obelix[1] > grep "unix is easy"  < myfile1  > myfile2
```

  - Write all lines of myfile1 containing phrase unix is easy to myfile2
- wc:
  - Count the number of chars/words/lines on stdin
  - Write the resulting statistics to stdout
- sort:
  - Sort all the input lines in alphabetical order and write to the standard output.

Pipes

- The pipe:
  - Connects stdout of one program with stdin of another
  - General form:

    ```
    command1 | command2
    ```

    - Stdout of command1 used as Stdin for command2
  - Example:

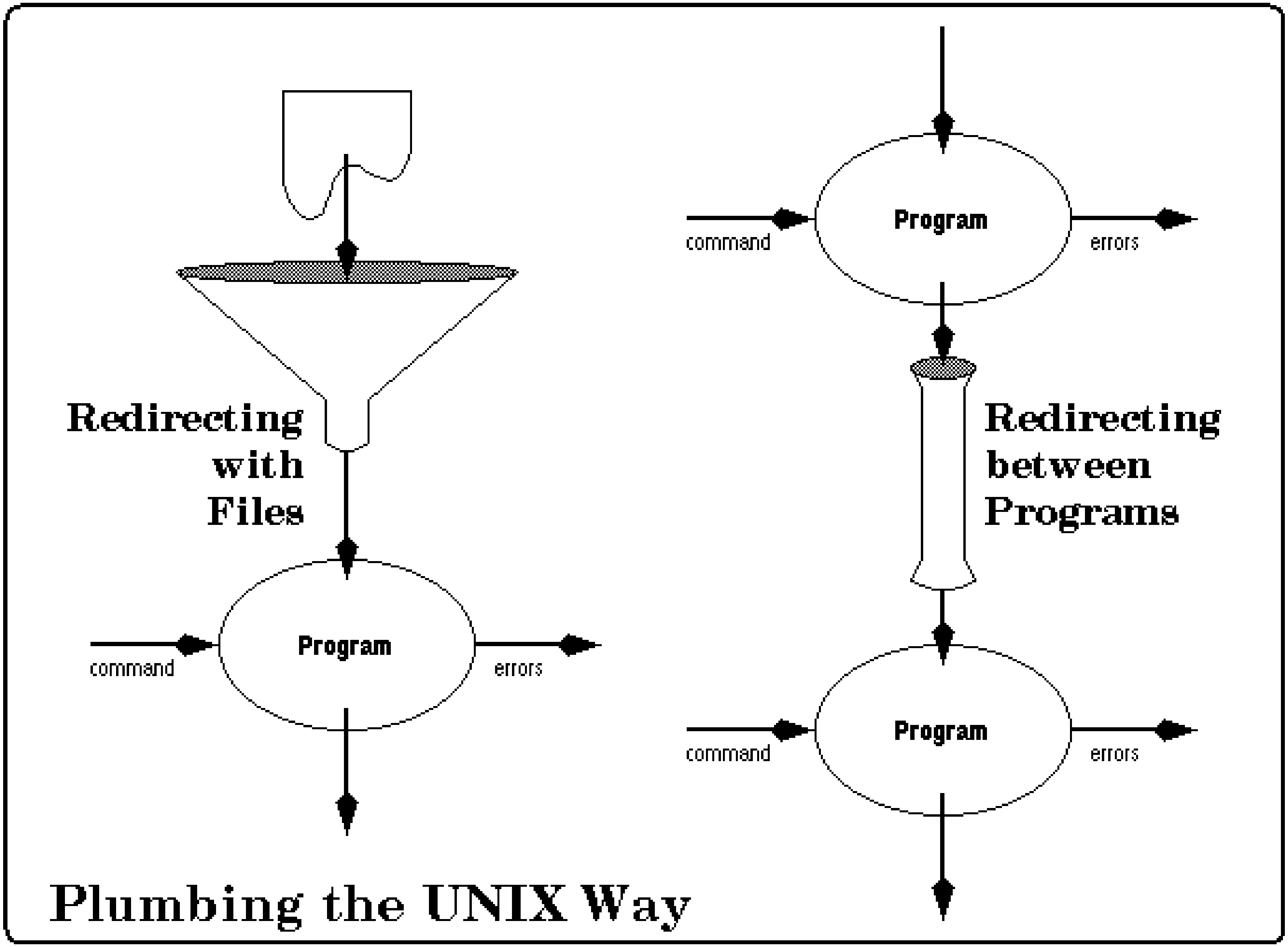
    ```
    obelix[1] > cat readme.txt | grep unix | wc -l
    ```

- An alternative way (not efficient) is to:

  ```
  obelix[2] > grep unix < readme.txt > tmp
  obelix[3] > wc -l < tmp
  ```

- Can also pipe stderr: `command1 |& command2`

Redirecting and Pipes (1)



Redirecting and Pipes (2)

- Note: The name of a command always comes first on the line.
- There may be a tendency to say:

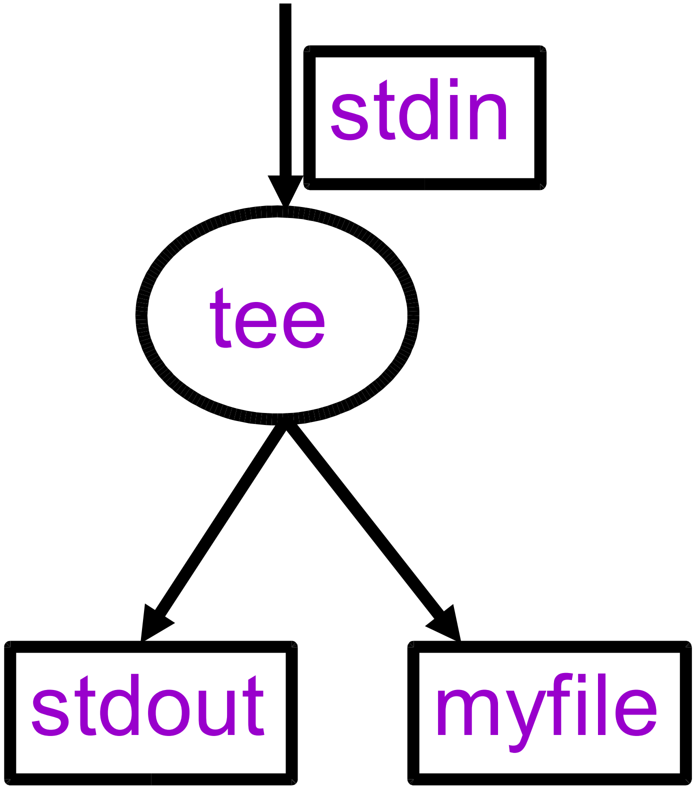
  ```
  obelix[1] > readme.txt > grep unix | wc -l
  ```

  - This is WRONG!!!
  - Your shell will go looking for a program named readme.txt
- To do it correctly, many alternatives!

  ```
  obelix[1] > cat readme.txt | grep unix | wc -l
  obelix[2] > grep unix < readme.txt | wc -l
  obelix[3] > grep unix readme.txt | wc -l
  obelix[4] > grep -c unix readme.txt
  ```

The tee Command

- tee - replicate the standard output
  - cat readme.txt | tee myfile



Devices and Redirection

- **Devices:** Unix lets you access a device (like a soundcard, or mouse, or ...) like it is a file; here are some Examples:

– /dev/hda1	first data partition on hard-disk A (B, C & D)
– /dev/hda	the boot sector of hard-disk A
– /dev/mouse	usually linked from a bus, Usb or serial device
– /dev/apm_bios	power management
– /dev/modem	for dial-up; usually is a link to a serial port
– /dev/isdn	for isdn dial-up
– /dev/ppp	for ppp connections
– /dev/cua0	a modem-friendly serial port
– /dev/ttyS0	a generic serical port
– /dev/cdrom	probably links to /dev/hdc
– /dev/fd0	a floppy – probably links to a format-spedific device

Regular Expressions

Regular Expressions

- A regular expression is a pattern which matches some regular (predictable) text.
- Regular expressions are used in many Unix utilities.
  - like grep, sed, vi, emacs, awk, ...
- The form of a regular expression:
  - It can be plain text ...

    > grep unix file (matches all the appearances of unix)
  - It can also be special text ...

    > grep ‘[uU]nix’ file (matches unix and Unix)

Regular Expressions and File Wildcarding

- Regular expressions are different from file name wildcards.
  - Regular expressions are interpreted and matched by special utilities (such as grep).
  - File name wildcards are interpreted and matched by shells.
  - They have different wildcarding systems.
  - File wildcarding takes place first!

```
obelix[1] > grep ‘[uU]nix’ file
obelix[2] > grep [uU]nix file
```

Regular Expression Wildcards

- A dot . matches any single character

  a.b matches axb, a$b, abb, a.b

  but does not match ab, axxb, a$bccb

- * matches zero or more occurrences of the previous single character pattern

  a*b matches b, ab, aab, aaab, aaaab, …

  but doesn’t match axb

- What does the following match?

  .*

Character Ranges

- Matching a set or range of characters is done with [...]
  - [wxyz] - match any of wxyz
    [u-z] - match a character in range u - z
- Combine this with * to match repeated sets
  - Example: [aeiou]* - match any number of vowels
- Wildcards lose their specialness inside [...]
  - If the first character inside the [...] is], it loses its specialness as well
  - Example: '[])}]' matches any of those closing brackets

Match Parts of a Line

- Match beginning of line with ^ (caret)

  ^TITLE

  - matches any line containing TITLE at the beginning
  - ^ is only special if it is at the beginning of a regular expression
- Match the end of a line with a $ (dollar sign)

  FINI$

  - matches any line ending in the phrase FINI
  - $ is only special at the end of a regular expression
- What does the following match?   ^WHOLE$

Matching Parts of Words

- Regular expressions have a concept of a “word” which is a little different than an English word.
  - A word is a pattern containing only letters, digits, and underscores (_)
- Match beginning of a word with **\<**
  - \<Fo matches Fo if it appears at the beginning of a word
- Match the end of a word with \>
  - ox\> matches ox if it appears at the end of a word
- Whole words can be matched too: \<Fox\>

More Regular Expressions

- Matching the complement of a set by using the ^
  - [^aeiou] - matches any non-vowel
  - ^[^a-z]*$ - matches any line containing no lower case letters
- Regular expression escapes
  - Use the \ (backslash) to "escape" the special meaning of wildcards
    - CA*Net
    - This is a full sentence\.
    - array\[3]
    - C:\\DOS
    - \[.*\]

Regular Expressions Recall

- A way to refer to the most recent match
- To remember portions of regular expressions
  - Surround them with \(...\)
  - Recall the remembered portion with \n where n is 1-9
    - ❖Example: '^\([a-z]\)\1'
      - matches lines beginning with a pair of duplicate (identical) letters
    - ❖Example: '^.*\([a-z]*\).*\1.*\1'
      - matches lines containing at least three copies of something which consists of lower case letters

Matching Specific Numbers of Repeats

- X\{m,n\} matches m -- n repeats of the one character regular expression X
  - E.g. [a-z]\{2,10\} matches all sequences of 2 to 10 lower case letters
- X\{m\} matches exactly m repeats of the one character regular expression X
  - E.g. #\{23\} matches 23 #s
- X\{m,\} matches at least m repeats of the one character regular expression X
  - E.g. ^[aeiou]\{2,\} matches at least 2 vowels in a row at the beginning of a line
- .\{1,\} matches more than 0 characters

Regular Expression Examples (1)

- How many words in /usr/dict/words end in ing?
  - grep -c 'ing$' /usr/dict/words

The -c option says to count the number of matches

- How many words in /usr/dict/words start with un and end with g?
  - grep -c '^un.*g$' /usr/dict/words
- How many words in /usr/dict/words begin with a vowel?
  - grep -ic '^[aeiou]' /usr/dict/words

The -i option says to ignore case distinction

Regular Expression Examples (2)

- How many words in /usr/dict/words have triple letters in them?

  -
    ```
    grep -ic '\(.\)\1\1' /usr/dict/words
    ```

- How many words in /usr/dict/words start and end with the same 3 letters?

  -
    ```
    grep -c '^\(...\).*\1$' /usr/dict/words
    ```

- How many words in /usr/dict/words contain runs of 4 consonants?

  -
    ```
    grep -ic '[^aeiou]\{4\}' /usr/dict/words
    ```

Regular Expression Examples (3)

- What are the 5 letter palindromes present in /usr/dict/words?

  -
    ```
    grep -ic '^\(.\)\(.\).\2\1$' /usr/dict/words
    ```

- How many words of the words in /usr/dict/words with y as their only vowel

  -
    ```
    grep '^[^aAeEiIoOuU]*$' /usr/dict/words | grep -ci 'y'
    ```

- How many words in /usr/dict/words do not start and end with the same 3 letters?

  -
    ```
    grep -ivc '^\(...\).*\1$' /usr/dict/words
    ```

Shell Environments

The Shell Environment

- Shell environment
  - Consists of a set of variables with values.
  - These values are important information for the shell and the programs run from the shell.
    - Example: PATH determines where the shell looks for the file corresponding to your command.
    - Example: SHELL indicates what kind of shell you are using.
  - You can define new variables and change the values of the variables.

Shell Variables (1)

- Shell variables are used by putting a $ in front of their names
  - e.g. echo $HOME
- Many are defined in .cshrc and .login
- Two kinds of shell variables:
  - Environment variables
    - available in the current shell and the programs invoked from the shell
  - Regular shell variables
    - not available in programs invoked from this shell (including “child” shells!)

Shell Variables (2)

- Setting Environment Variables:
  - export varname=varvalue
- Example:

```
obelix[1] > export myvar="unix is easy"
obelix[2] > echo myvar
myvar
obelix[3] > echo $myvar
unix is easy
```

- Clearing out regular variables:

```
obelix[4] > unset myvar
obelix[5] > echo $myvar
myvar: undefined variable
```

Shell Variables (3)

- Example (in bash):

```
obelix[3] > MYVAR1="Unix is easy"    (Reg. Variable)
obelix[4] > export MYVAR2="Windows is easier"
obelix[6] > echo $MYVAR1
Unix is easy
```

(Now start a new shell:)

```
obelix[5] > bash
obelix[6] > echo $MYVAR1

obelix[7] > echo $MYVAR2
Windows is easier
```

Shell Vairables (4)

- Common shell variables:
  - SHELL: the name of the shell being used
  - PATH: where to find executables to execute
  - MANPATH: where man looks for man pages
  - LD_LIBRARY_PATH: where libraries for executables are found at run time
  - USER: the user name of the user logged in
  - HOME: the user’s home directory
  - TERM: the kind of terminal the user is using
  - DISPLAY: where X program windows are shown
  - HOST: the name of the host logged on to
  - REMOTEHOST: the name of the host logged in from

More on Unix Quoting

- Single Quotes '...'
  - Stop variable expansion ($HOME, etc.)

```
obelix[16] > echo “Welcome $HOME”
Welcome /home/s1/student/1991/katleen/
obelix[17] > echo ‘Welcome $HOME’
Welcome $HOME
```

- Back Quotes \`…\`
  - Replace the quotes with the results of the execution of the command.
  - E.g. ``obelix[18] > export prompt=`hostname` ``

The Search Path

- How does Unix find commands to execute?
  - If you specify a pathname, the shell looks into that path for the executable.
  - If you specify a filename, (without / in the name), the shell looks for it in the search path.
  - There is a variable PATH or path

    ```
    obelix[1] > echo $PATH
    /home/s1/student/1991/katleen/bin:/bin:/usr/local/bin:.
    ```
- The shell does not look for executables in your current directory unless:
  - You specify it explicitly, e.g. ./a.out
  - . is specified in the path variable

Selecting Different Versions of a Command

- There may be multiple versions of the same command in your search path.

```
obelix[1] > whereis ps
ps: /usr/bin/ps /usr/ucb/ps
```

- The shell searches in each directory of the $PATH in left to right order and executes the first version.

```
obelix[2] which ps
/usr/bin/ps
obelix[3] /usr/ucb/ps
```

Shell Startup

- When csh and tcsh are executed, they run certain configuration files:
  - .login run once when you log in
    - Contains one-time things like terminal setup.
  - .cshrc run each time another [t]csh process runs
    - Sets lots of variables, like PATH.
- Other shells such as sh use a different file, like .profile to do similar things.
- Only modify the lines that you fully understand!

The alias Command

- alias format:
  - alias   alias-name   real-command
    - ❖ alias-name is one word
    - ❖ real-command can have spaces in it
- Any reference to alias-name invokes real-command. Some examples:
  - alias rm rm –i
  - alias cp cp –i
  - alias mv mv –i
  - alias ls /usr/bin/ls –CF
    - ❖ This shows us the /, *, @ after file names using ls.
- Put aliases in your .bashrc file to set them up whenever you log in to the system!

Command History (1)

- obelix[9] > history

```
1  10:57   emacs
2  10:57   ls -l .cshrc
3  10:57   cp .cshrc .cshrc2
4  10:57   emacs .cshrc
5  11:01   ps
6  13:46   pwd
7  13:46   cd ..
8  13:46   pine
9  13:46   history
```

Command History (2)

- You can rerun a command line in the history
  - !!   reruns last shell command
  - !str rerun the latest command beginning with str
  - !n   (where n is a number) rerun command number n in the history list
- tcsh allows you to use arrow keys to wander the history list easily.
- The length of the history list is determined by the variable history, likely set in your .cshrc file.

```
set history = 40
```

- The variable savehist determines how much history to save in the file named in histfile for your next session; these are also likely set in your .bashrc file.

Command and Filename Completion

- In tcsh and bash, you can let the shell complete a long command name by:
  - Typing a prefix of the command.
  - Hitting the TAB key.
  - The shell will fill in the rest for you, if possible.
- tcsh and bash also complete file names:
  - Type first part of file name.
  - Hit the TAB key.
  - The shell will complete the rest, if possible.
- Difference:
  - First word: command completion.
  - Other words: file name completion.