today

mon 30 sep 2002

- homework #2 — EXTENSION — due MON OCT 7, 6AM
- see class web page for hw#2 script instructions
- shell scripts

references for these slides:
— cs3995 Spring 2002 lecture notes, by Henning Schulzrinne
— UNIX Power Tools, by Peek, O'Reilly and Loukides

scripting languages (1)

- not a well-defined term
- derived from shell (command-line) scripts
- often typed directly by user
- usually no compile-link-run cycle, but interpreted or compiled (i.e., just in time — JIT)
- we'll look at typical examples:
  - sh, bash (today)
  - sed — pattern editing (later)
  - awk — pattern language (later)
  - perl — a real language for string processing (later)
  - Tcl — shell-like, easy extensible, graphical GUI (later)
  - python — object-oriented (later)

scripting languages (2)

- often loosely typed
  - no explicit variable and type declaration
  - variables treated as strings or numbers according to context
- dynamic memory allocation with automatic garbage collection
- text processing:
  - regular expressions
  - sorting
  - other utilities...
- procedural, but often with object-oriented extensions
- some are derived from substitution instead of evaluation — Tcl, sh
- some allow mixed-language programming — Tcl, perl, python

shells

- each OS has one, but there are different levels of sophistication:
  - Windows: DOS command prompt
  - UNIX:
    - sh — Bourne shell, the original /bin/sh
      - bash — Bourne-Again Shell, derived from sh
      - ksh — Korn shell = superset of sh
    - csh — with C-like syntax
      - tcsh — improved version of csh

sh (1)

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

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


sh (2)

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

sh (3)

- are you runnning the Bourne shell?
  - type `sh$ echo $SHELL`
  - if the answer is `/bin/sh`, then you are
  - if the answer is `/bin/bash`, then that's close enough
  - otherwise, you can start the Bourne shell by typing `sh` at the UNIX prompt
  - enter `Ctrl-D` or `exit` to exit the Bourne shell and go back to whatever shell you were running before...

sh (4)

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

sh (5)

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

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

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

(note that `sh$` *means the unix prompt, i.e.,* `unix$`*)*

sh (6)

- quoting
  - `'something'`: preserve literally
  - `"something"`: allow $ variable expansion
  - `$ 'escape-sequence'`: e.g., `$'\a'`
- comments
  - single line comments only
  - line begins with # character

sh (7) — simple commands

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

sh (8) — pipes

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

```
sh$ echo "hello world" | wc -w
2
```

sh (9) — shell variables

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

sh (10) — environment variables

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

```
sh$ echo $TERM
vt100
```


sh (11) — looping constructs

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

sh (12) — loop examples

- while

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

- for

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

sh (13) — if

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


sh (14) — case

- example:

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

- pattern matching:
  - `?)` matches a string with exactly one character
  - `?*)` matches a string with one or more characters
  - `[yY]|[yY][eE][sS])` matches `y, Y, yes, YES, yES`...
  - `/*/*[0-9])` matches filename with wildcards like `/xxx/yyy/zzz3`
  - `'does it match?'` matches "does it match?" — with quote, shell doesn't ignore spaces or interpret ?
  - `"$match")` matches the text `match`

sh (15) — expansion

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

sh (16) — brace expansion

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

```
sh$ echo a{d,c,b}e
ade ace abe
```

- useful for generating list of filenames:

```
sh$ mkdir hw{1,2,3}
sh$ ls
hw1  hw2  hw3
```

sh (17) — tilde expansion

- expands to `$HOME`
- examples: `cs3157` ⇒ `/u/5/c/cs3157`
  `/html` ⇒ `/home/sklar/html`

sh (18) — command substitution

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

```
x=ls
echo ‘ls‘
```

- danger! don’t mess with built-in command names...

sh (19) — filename expansion

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

sh (20) — redirections

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

built-in commands (1)

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

built-in commands (2)

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

built-in commands (3)

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