Crash Course on UNIX and Systems Tools

Day 3 --- More Scripting and Project Structure

Correction from Day 2

“How do you **`grep`** two words at once?” --- Want to correct my previous answer and emphasize **regular expressions**

- Don’t use `[]` --- that specifies a **range** of values (more later)
- **`grep -e “DGRAM\|STREAM”`** --- need to escape “|”
- **`grep -E “DGRAM|STREAM”`** --- no need to escape “|”
- **`grep -E “DGRAM.*STREAM|STREAM.*DGRAM”`**
  - **IFF** you’re searching for words **on the same line**

Correction from Day 2

Usage statement convention --- for **required** arguments, you **don’t** need braces “ **{ }** ” to specify.

I’ll upload a correct version of scripts to the course drive.

Scripting: *Text editors (cont. from Day 2)*

- **`vim`** configuration is a great way to customize your text editor
  - Should be called **`~/.vimrc`** (create a file if nonexistent)

```
$ touch ~/.vimrc ; vim ~/.vimrc
```

Scripting: *Text editors (cont. from Day 2)*

Some nice configurations:

- **`set shiftwidth = X`** : Shifting “**>>**” and “**<<**” to X spaces
- **`set tabstop = X`** : Setting tab to X spaces
- **`set expandtab`** : Tab to be processed as spaces
- **`set number`** : Set line numbers

An awesome config from Simone Campanoni (CS Prof @ NU):
https://github.com/scampanoni/vim_configuration

Exercises: *Question (cont. from Day 2)*

Can you check how many sockets (according to **`netstat`**) are of type **`DGRAM`** *every 5 seconds*? ***

```
#!/bin/bash

# Vetting --- SEE pedantic script for documented
# explanations about if statments (or watch the video)
if [ -z "$1" ] ; then
    echo "USAGE: ./myscript { OUTFILE } " ;
    exit 1 ;
elif [ -f "$1" ] ; then
    echo -e "ERROR: File \"${1}\" already exists" ;
    exit 1 ;
fi


OUT="$1";
```

```
# Add SECONDS to usage statement
echo "USAGE: ./myscript OUTFILE [SECONDS]" ;
```

```
OUT="$1";

# Setting the TIME variable to either the
# optional second input or 1 (for 1s). The
# parameter is vetted to be an integer using
# a regular expression statement
if [ -z "$2" ] ; then
    TIME=1 ;
elif ! [[ "$2" =~ ^[1-9][0-9]*$ ]] ; then
    echo -e "ERROR: Please specify a positve integer for SECONDS!" ;
    exit 1 ;
else
    TIME="$2" ;
fi
```

Useful guide to regular expressions:
https://medium.com/factory-mind/regex-tutorial-a-simple-cheatsheet-by-examples-649dc1c3f285 , https://tldp.org/LDP/GNU-Linux-Tools-Summary/html/x11655.htm

Environments: *Setting PATH*

`export`

- Marks an environment variable to be visible or accessible in spawned child processes in the shell
- If we want to run our script from anywhere in the shell, we should add it to **`PATH`**

Environments: *Setting PATH*

```
$ mkdir -p bin ; mv myscript bin/ ; cd ./bin
$ export PATH=`pwd`:$PATH
$ echo $PATH
...
```

Environments: *Executing in the current shell*

`source`

- When executing a program, it starts in a **child** shell process --- running **`source`** will execute the program in the **current** shell

Environments: *Executing in the current shell*

```
$ cat > test
export PATH=1 --- very silly
^D
$ chmod +x test ; ./test ; echo $PATH
...
$ source ./test ; echo $PATH
1
$
```

Environments: *Bash configuration*

`~/.bashrc`

- **`~/.bashrc`** is the bash configuration file --- it is sourced at the start of each interactive shell
- You can customize your bash environment
  - Add **`alias`**es
  - Add commands to execute before the prompt

Project Structure: *Structuring your code*

We'll start with some code to look at for this section of the workshop

```
wget --no-check-certificate
'https://docs.google.com/uc?export=download&id=1V
pcAo17lYa1HONtVG1ix7efdQzXwdYvD' -O lib.c
```

Project Structure: *Compilation*

Let’s compile it! Many of you have seen basic commands to compile code on the command line to generate an executable

- We’ll use **`gcc`** --- the most commonly used C compiler

```
$ gcc lib.c
$ ls
a.out*  lib.c
```

Project Structure: *Compilation*

Some tools to examine what you've generated:

- **`file`** --- Determines the file type of your input
- **`ldd`** --- Lists the dependences of shared libraries (like **`libc`**) of your executable
- **`objdump`** --- Info (symbols, assembly output, etc.) on object files
  - Won't look at this but *very* useful ; you'll encounter it in 213 and elsewhere
  - Useful flags: **`-d`** (disassemble), **`-t`** (fetch symbols)

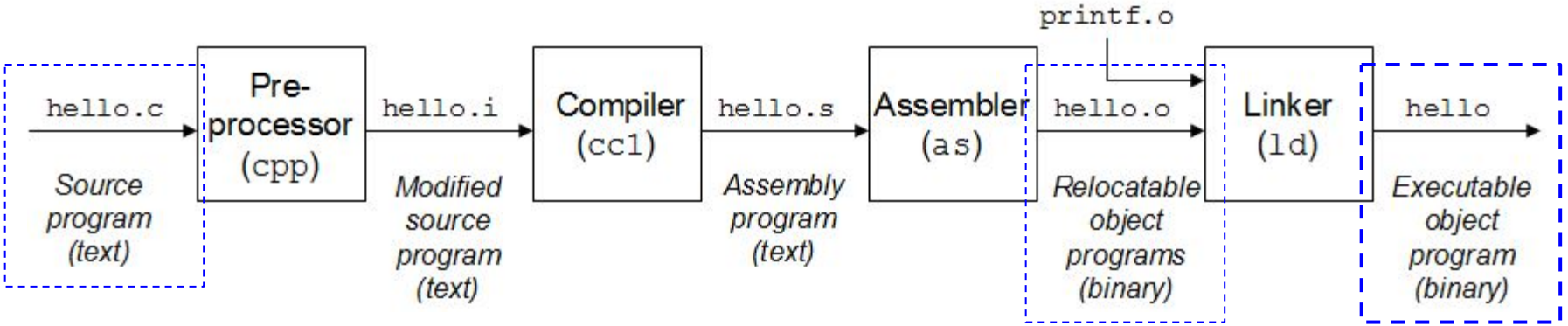
Project Structure: *Compilation*

```
$ file a.out
a.out: ELF 64-bit LSB shared object, x86-64,
version 1 (SYSV), dynamically linked …

$ ldd a.out
        ...
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
        ...
```

Project Structure: *Compilation*

- Generally speaking, the simple invocation **`gcc`** will walk through these steps:



Project Structure: *Compilation*

- An idea of what the compiler invokes (FYI):

```
gcc -v lib.c
```

- To specify an **output** file (instead of the default **`a.out`**)

```
gcc lib.c -o lib
```

- To turn on all (optional) **warnings**, very useful in in practice

```
gcc -Wall lib.c -o lib
```

Project Structure: *Structuring your code*

- Header files (`.h`)
  - Essential for **defining the interface** for your application
  - Important for declarations, global definitions, etc. that both the compiler and user can use, etc.

- The implementation (in `.c`) should be **separate** from the **interface** AND separate from the **users** of the code
  - Separating test cases from the library