CSC/CEG 3150 Tutorial

Programming Tips and Tools for Assignment 3

XIAO Zigang
zgxiao@cse.cuhk.edu.hk

Department of Computer Science and Engineering
The Chinese University of Hong Kong

November 4, 2008

Outline

1. Outline
2. Tips
   - `getopt`
   - `xxd`
   - `dd`
   - `mkfs`
   - `[u]mount`
   - `endian`
   - `others`
3. Q & A
4. Acknowledgement

Command line argument

- Recall that for some command in Linux,e.g. `/bin/ls`
  - `/bin/ls -l -a`
  - `/bin/ls -la`
  - `/bin/ls -la --file-type --color=auto -w 30 /usr /lib`
- We can observe at least three behaviors:
  - It receives multiple arguments
  - There are short arguments and long arguments, which may require a following value or not
  - We can group similar short arguments together

How to handle argument elegantly?

- Different milestones in Assignment 3 need different input arguments
  - `./Main -i`
  - `./Main -r`
  - `./Main -w`
  - `./Main -a`
  - `./Main -r`
- How do we parse the command-line argument?
  - Hard code it with the help of `strtok`
    - Tedious, different program needs similar parse code
  - We do not want those stuff bother program's main logic
    - Is there any standard way to do it? - `getopt`

argc and argv revisited

- `int argc` : Number of arguments
  - Including the executable name
- `char * argv[]` : List of arguments
  - Null terminated.
- E.g. `./main -a test -b`
  - `argc = 4`
  - `argv[0] = ./Main`
  - `argv[1] = -a`
  - `argv[2] = test`
  - `argv[3] = -b`
- `argv[4](=0)` is NULL.

argc and argv example

```
#include <stdio.h>
int main(int argc,char * argv[]){
    int arg;
    for(arg = 0; arg < argc ; arg++){
        if(’-’ == argv[arg][0] )
            printf("Options: %s\n", argv[arg]+1);
        else
            printf("argument %d: %s\n", arg,argv[arg]);
    }
}
```

Listing 1: Use argc and argv to parse arguments

```
$ ./args -i -lr ’hi there’ -f main.c
argument 0: args
option: i
option: lr
argument 3: hi there
option: f
argument 5: main.c
```

Listing 2: sample output

getopt

```
#include <unistd.h>
int getopt(int argc, char * const argv[],const char * optstring);
extern char * optarg;
extern int optind, opterr, optopt;
```

Listing 3: prototype of getopt

- getopt uses argc,argv as parameters, as well as a string **“optstring”**
- optstring tells getopt what options should be handled, and which option should follow a value
- If a character is followed by a colon, that means the corresponding option requires an argument
    - `optstring = "ab"` means your program accepts `"-a"` and `"-b"`
    - `optstring = "a:b"` means your program accepts `"-a"` and `"-b"`, and option a requies an argument, which will be stored in `"optarg"`

Example of getopt

```c
#include <unistd.h>
#include <stdio.h>

int main(int argc,char * argv[]){
        int opt;
        while( (opt = getopt(argc,argv,"if:lr")) != -1){
                switch(opt){
                case 'i':
                case 'l':
                case 'r':
                        printf("Option: %c\n",opt);
                        break;
                case 'f':
                        printf("filename: %s\n",optarg);
                        break;
                case ':':
                        printf("Option needs a value\n");
                        break;
                case '?':
                        printf("Unknown option: %c\n",optopt);
                        break;
                }
        }
        for(; optind < argc; optind++) /* note this */
                printf("argument: %s\n",argv[optind]);
        return 0;
}
```

Listing 4: Rewritting parser

A few words about `extern`

- *declare* a variable:
  - `int a;`
- *define* a variable:
  - `int a=0;`
- In functions, the variables we *defined* are allocated in stack.
  - Automatically disappears after function returns.
- The *external variables* are globally accessible.
- *"If the program is in several source files, and a variable is defined in file1 and used in file2 and file3, then extern declarations are needed in file2 and file3 to connect the occurrences of the variable."* [1]
- `optarg,optind,etc.` are defined in `<getopt.h>`
- You may check it in `/usr/include/getopt.h`

[1] K&R, The C Programming Language

Some important facts...

- If an option needs an extra argument, `optarg` associates with it.
- When all the options are processed, `getopt` returns -1
- When an unrecognized option is received, `getopt` returns `'?'`,`optarg` saves this option character
- Actually `getopt` rewrites `argv` array, see code linu 22-23 in Listing 4
- `getopt_long` can handle long arguments
- For more details, ask `man`
  - `man getopt`
  - `man getopt_long`

xxd

- Creates a hex dump of a given file or standard input
- In Assignment 3
  - Try `xxd /dev/ram` - Too much output
  - Try `xxd -s 0x40 -l 160 /dev/ram`
    - Start to read at byte 0x40 (= byte 64)
    - Read 160 bytes
- The right column of the printout contains ascii output.
  - `man ascii` to get more detail
- Another tool **hd** can do similar work

```
0000040: 0000 29e7 41f7 4820 2020 2020 2020 2020  ..).A.H
0000050: 2020 4641 5433 3220 2020 0e1f be77 7cac    FAT32   ...w|.
0000060: 22c0 740b 56b4 0ebb 0700 cd10 5eeb f032  ".t.V.......^..2
0000070: e4cd 16cd 19eb fe54 6869 7320 6973 206e  .......This is n
0000080: 6f74 2061 2062 6f6f 7461 626c 6520 6469  ot a bootable di
0000090: 736b 2e20 2050 6c65 6173 6520 696e 7365  sk.  Please inse
00000a0: 7274 2061 2062 6f6f 7461 626c 6520 666c  rt a bootable fl
00000b0: 6f70 7079 2061 6e64 0d0a 7072 6573 7320  oppy and..press
00000c0: 616e 7920 6b65 7920 746f 2074 7279 2061  any key to try a
00000d0: 6761 696e 202e 2e2e 200d 0a00 0000 0000  gain ... .......
```

Listing 5: Sample output of xxd

data definition[2]

- dd is a common UNIX program whose primary purpose is the **low-level** copying and conversion of **raw data**
- Like most well-behaved commands, dd reads from its standard input and writes to its standard output
- Use `if` and `of` to specify input and output
- Use `bs` and `count` to specify data size$(= bs * count)$
- Example
  - Copies from /dev/zero to ./zeros with 2 blocks, each block counts for 1k bytes.

    ```
    $dd if=/dev/zero of=./zeros bs=1k count=2
    ```
  - Copies from /dev/ram0 to ./header with a 512-byte block.

    ```
    $dd if=/dev/ram0 of=./header bs=512 count=1
    ```

[2] History of dd:`http://www.catb.org/jargon/html/D/dd.html`

mkfs and mkfs.vfat utilities

- Used to build a file system on a device
- It is just a wrapper for specific file system maker, e.g. mkfs.vfat links to mkdosfs
- Example:

```
/sbin/mkfs.vfat -v -F 32 -f 2 -S 512 -s 1 -R 32 /dev/ram0
```

Formats /dev/ram0 to a FAT32 file system

`-F`	`specify FAT size (12,16,32)`
`-f`	`Fat table number`
`-S`	`logical sector size, default is 512B`
`-s`	`sector number per cluster`
`-R`	`number of reserved sectors`

Play with disk image

- All things in UNIX are files...
- You may use the **ramdisk** `/dev/ram[0-9]` to create FAT32 file system [3]
  - ramdisk is memory mapping to the file, so it is fast
- You may also use a normal file to create FAT32 file system
-
  ```
  # touch a 64MB file
  $ dd if=/dev/zero of=./fs bs=1M count=64
  # format as a FAT32 file system
  $ mkfs.vfat ./fs
  # Check the signature 0x55 and 0xAA
  $ xxd -s 510 -l 22 ./fs
  ```

[3] (Ubuntu user might need **sudo** to access them)

Mounting the disk image

- We have disk image now, so what?

[4] A loop device makes a file accessible as a block device, e.g. hard disk.

Mounting the disk image

- We have disk image now, so what?
- We can mount it to our file system (hierarchy)

[4] A loop device makes a file accessible as a block device, e.g. hard disk.

Mounting the disk image

- We have disk image now, so what?
- We can mount it to our file system (hierarchy)
- `$ mkdir /mnt/rd`

  `$ mount -t vfat -o loop,umask=000 /dev/ram0 /mnt/rd` [4]

[4] A loop device makes a file accessible as a block device, e.g. hard disk.

Mounting the disk image

- We have disk image now, so what?
- We can mount it to our file system (hierarchy)
-
  ```
  $ mkdir /mnt/rd
  $ mount -t vfat -o loop,umask=000 /dev/ram0 /mnt/rd
  ```
  [4]
- Use `fdisk -l` to list the partition tables of devices in your machine(need root privilege)

[4] A loop device makes a file accessible as a block device, e.g. hard disk.

Syntax of mount

```
$ mount -t fstype -o options device dir
```

-t	file system type use ‘‘vfat’’ for FAT32
device	the source device(file) you want to mount
dir	the destination directory(mount point)
-o	options

Some useful options are:

umask	the bitmask of the permissions that are not present(000 means anyone can read & write)
loop	use a loop device

Mounting the disk image(cont.)

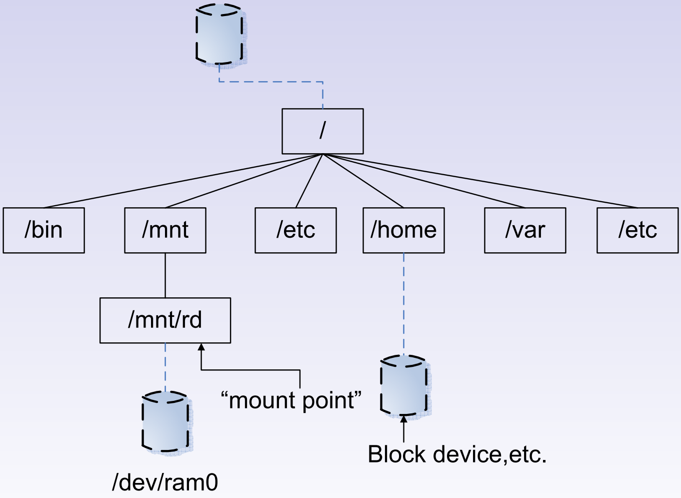

Figure: Filesystem Hierarchy

Unmounting the disk image

- We may do the reverse using `umount`
- The umount command detaches the file system(s) mentioned from the file hierarchy
- E.g. `/dev/sda1` is mounted on `/usr/local`, and we want to unmount it.
  - `$ umount /usr/local`
  - `$ umount /dev/sda1`
- The first way is more appropriate since device may be mounted on more than one directory

Big endian and little endian

- In x86, the data is written in little endian ordering
  - The least significant byte value is at the lowest address
  - The other bytes follow in increasing order of significance
  - E.g. Storage in memory with value = `0x0A0B0C0D`
- Numbers will appear in reverse order
- Strings will appear in the expected order

0D	0C	0B	0A

Low address → High address

Table: Illustration of endian

Example and codes

- What is the output?

```c
#include <stdio.h>

int main(){
        char ch[]={1,2,3,4};
        long int a=*((long int*)ch);
        // %.08x outputs a 8-digit hex number, padded with 0
        // man printf for more detail
        printf("0x%.08x\n",a);
        return 0;
}
```

Example and codes

- What is the output?

```c
#include <stdio.h>

int main(){
        char ch[]={1,2,3,4};
        long int a=*((long int*)ch);
        // %.08x outputs a 8-digit hex number, padded with 0
        // man printf for more detail
        printf("0x%.08x\n",a);
        return 0;
}
```

- Result in sparc machine(big endian): `0x01020304`

Example and codes

- What is the output?

```c
#include <stdio.h>

int main(){
        char ch[]={1,2,3,4};
        long int a=*((long int*)ch);
        // %.08x outputs a 8-digit hex number, padded with 0
        // man printf for more detail
        printf("0x%.08x\n",a);
        return 0;
}
```

- Result in sparc machine(big endian): 0x01020304
- Result in Pentium machine(little endian): 0x04030201

Example and codes

- What is the output?

```
#include <stdio.h>

int main(){
        char ch[]={1,2,3,4};
        long int a=*((long int*)ch);
        // %.08x outputs a 8-digit hex number, padded with 0
        // man printf for more detail
        printf("0x%.08x\n",a);
        return 0;
}
```

- Result in sparc machine(big endian): `0x01020304`
- Result in Pentium machine(little endian): `0x04030201`
- Check your processor
  - `$ uname -sp`
  - `$ cat /proc/cpuinfo` (Linux only)

Other things you have learned

- editor : vi & emacs & ...
- compiler : gcc
- debugger : printf & gdb
- make & makefile

Q & A Session

Thank You

Now you have enough handful tools to finish Assignment 3

- See You Next Week -

Acknowledgement

- Some materials and pictures are from last year's tutorial notes made by *Mr.Cheong Chi Hong*
- The style of this slide is adapted from the template made by *HUANG Zheng-hua* in *Wuhan University*

- Good Reference : cfaq
  `http://c-faq.com/`