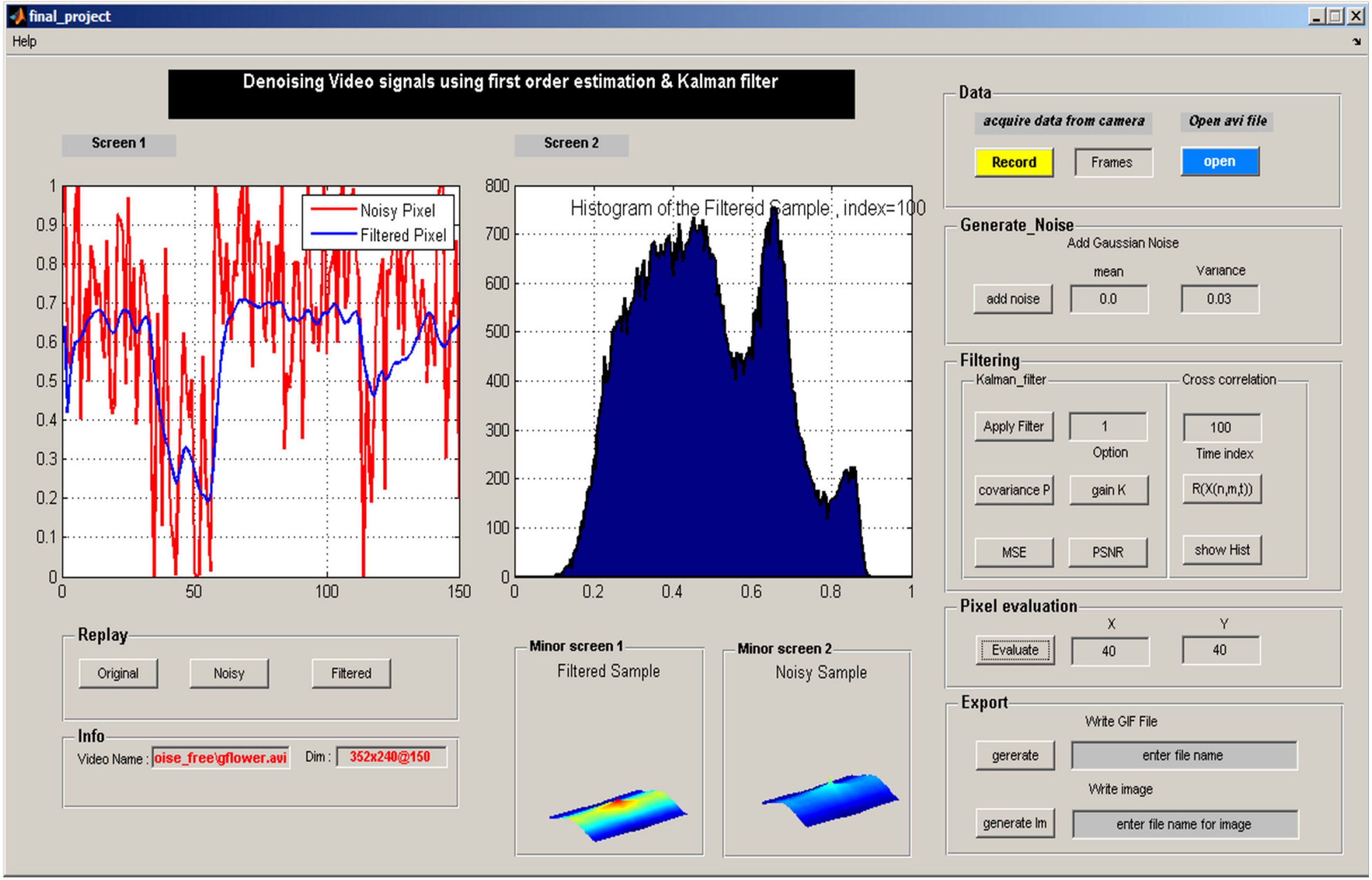
Introduction and Tools

Cheng-Hsin Hsu

National Tsing Hua University

Department of Computer Science

Parts of the course materials are courtesy of Prof. Chun-Ying Huang



Denoising Video signals using first order estimation & Kalman filter

Add Gaussian Noise

mean: 0.0

Variance: 0.03

Option: 1

Time index: 100

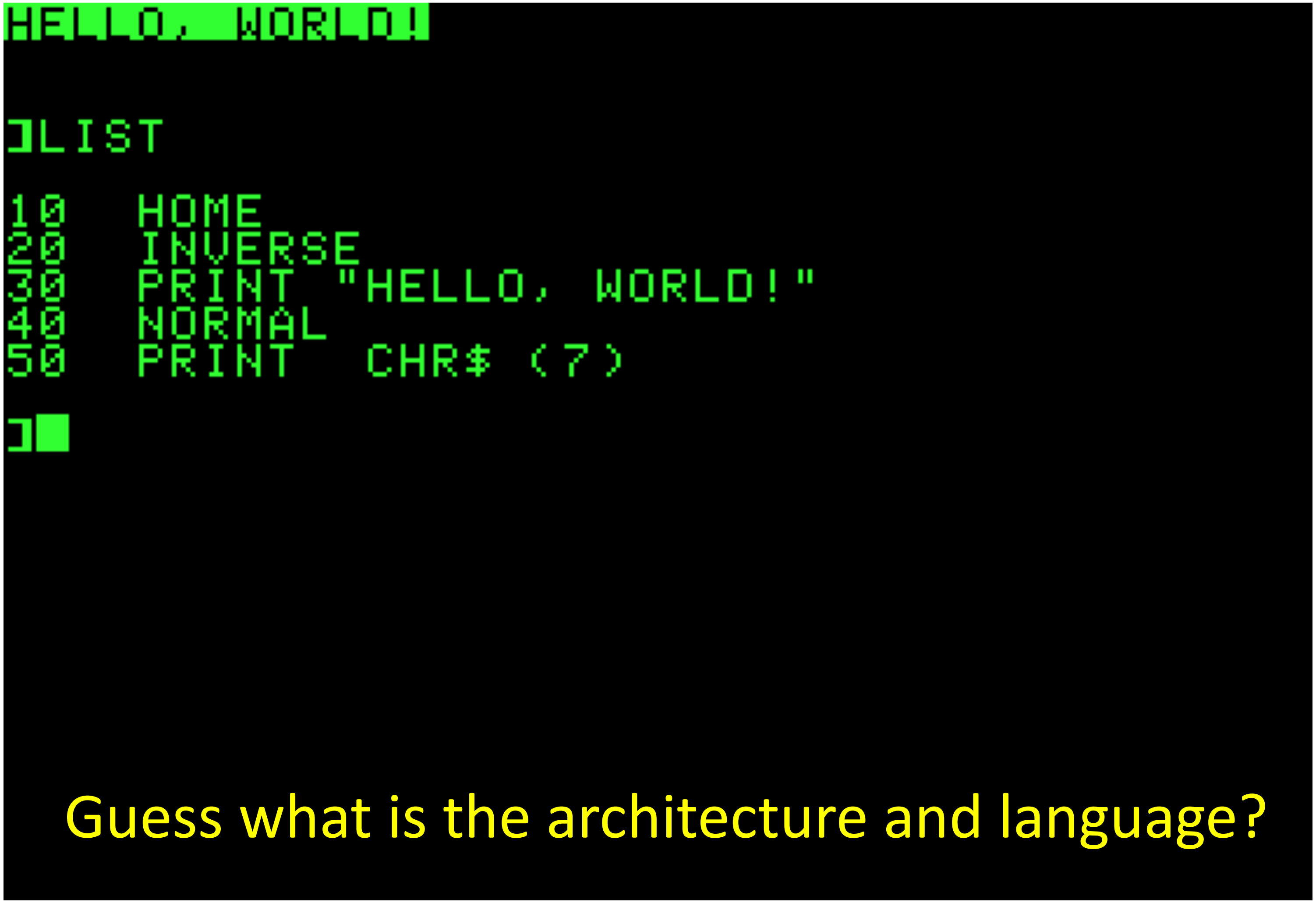
X: 40

Y: 40

Video Name : oise_free\gflower.avi

Dim : 352x240@150

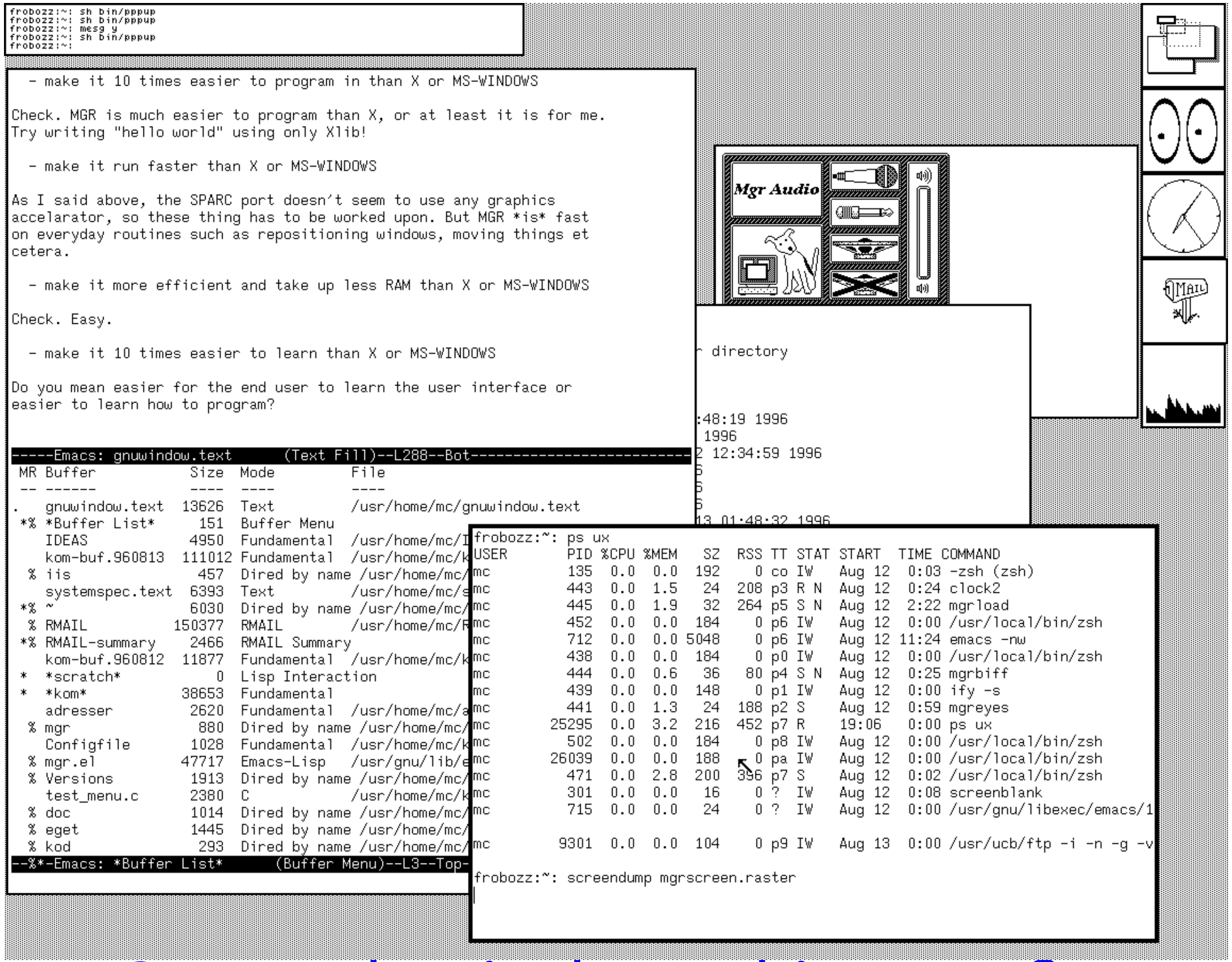
My First Computer



```
HELLO, WORLD!


]LIST

10   HOME
20   INVERSE
30   PRINT "HELLO, WORLD!"
40   NORMAL
50   PRINT  CHR$ (7)

]█
```

Guess what is the architecture and language?

My First GUI (All Utilities Run in XTerm)



```
frobozz:~: sh bin/pppup
frobozz:~: sh bin/pppup
frobozz:~: mesg y
frobozz:~: sh bin/pppup
frobozz:~:
```

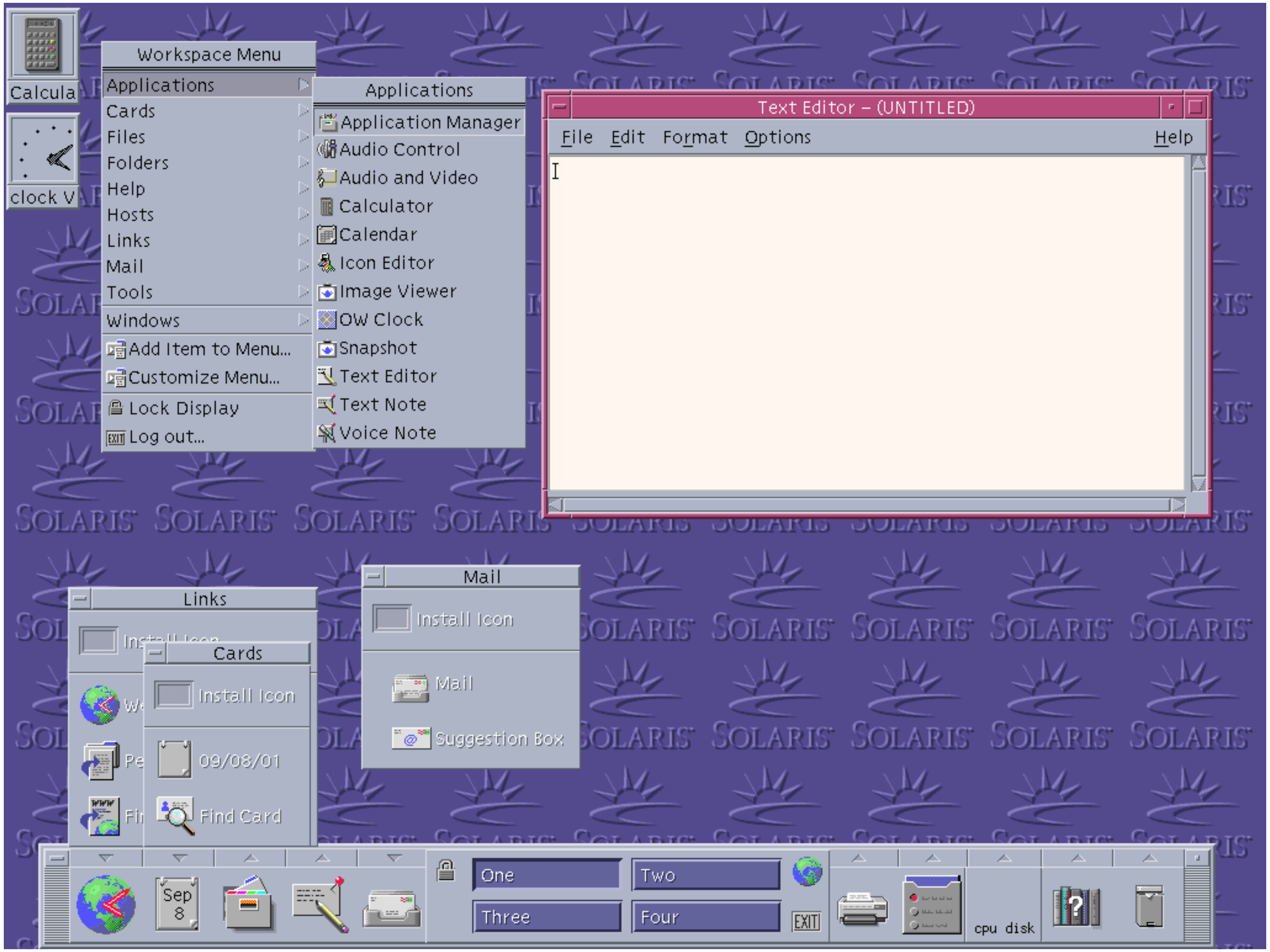
```
  - make it 10 times easier to program in than X or MS-WINDOWS

Check. MGR is much easier to program than X, or at least it is for me.
Try writing "hello world" using only Xlib!

  - make it run faster than X or MS-WINDOWS

As I said above, the SPARC port doesn't seem to use any graphics
accelarator, so these thing has to be worked upon. But MGR *is* fast
on everyday routines such as repositioning windows, moving things et
cetera.

  - make it more efficient and take up less RAM than X or MS-WINDOWS

Check. Easy.

  - make it 10 times easier to learn than X or MS-WINDOWS

Do you mean easier for the end user to learn the user interface or
easier to learn how to program?


-----Emacs: gnuwindow.text       (Text Fill)--L288--Bot--------------------------
 MR Buffer              Size  Mode          File
 -- ------              ----  ----          ----
.   gnuwindow.text     13626  Text          /usr/home/mc/gnuwindow.text
 *% *Buffer List*        151  Buffer Menu
    IDEAS               4950  Fundamental   /usr/home/mc/I
    kom-buf.960813    111012  Fundamental   /usr/home/mc/k
  % iis                  457  Dired by name /usr/home/mc/
    systemspec.text     6393  Text          /usr/home/mc/s
 *% ~                   6030  Dired by name /usr/home/mc/
  % RMAIL             150377  RMAIL         /usr/home/mc/R
 *% RMAIL-summary       2466  RMAIL Summary
    kom-buf.960812     11877  Fundamental   /usr/home/mc/k
 *  *scratch*              0  Lisp Interaction
 *  *kom*              38653  Fundamental
    adresser            2620  Fundamental   /usr/home/mc/a
  % mgr                  880  Dired by name /usr/home/mc/
    Configfile          1028  Fundamental   /usr/home/mc/k
  % mgr.el             47717  Emacs-Lisp    /usr/gnu/lib/e
  % Versions            1913  Dired by name /usr/home/mc/
    test_menu.c         2380  C             /usr/home/mc/k
  % doc                 1014  Dired by name /usr/home/mc/
  % eget                1445  Dired by name /usr/home/mc/
  % kod                  293  Dired by name /usr/home/mc/
--%*-Emacs: *Buffer List*      (Buffer Menu)--L3--Top-
```

```
r directory



:48:19 1996
 1996
2 12:34:59 1996
6
6
6
13 01:48:32 1996
```

```
frobozz:~: ps ux
USER        PID %CPU %MEM   SZ  RSS TT STAT START  TIME COMMAND
mc          135  0.0  0.0  192    0 co IW   Aug 12  0:03 -zsh (zsh)
mc          443  0.0  1.5   24  208 p3 R N  Aug 12  0:24 clock2
mc          445  0.0  1.9   32  264 p5 S N  Aug 12  2:22 mgrload
mc          452  0.0  0.0  184    0 p6 IW   Aug 12  0:00 /usr/local/bin/zsh
mc          712  0.0  0.0 5048    0 p6 IW   Aug 12 11:24 emacs -nw
mc          438  0.0  0.0  184    0 p0 IW   Aug 12  0:00 /usr/local/bin/zsh
mc          444  0.0  0.6   36   80 p4 S N  Aug 12  0:25 mgrbiff
mc          439  0.0  0.0  148    0 p1 IW   Aug 12  0:00 ify -s
mc          441  0.0  1.3   24  188 p2 S    Aug 12  0:59 mgreyes
mc        25295  0.0  3.2  216  452 p7 R    19:06   0:00 ps ux
mc          502  0.0  0.0  184    0 p8 IW   Aug 12  0:00 /usr/local/bin/zsh
mc        26039  0.0  0.0  188    0 pa IW   Aug 12  0:00 /usr/local/bin/zsh
mc          471  0.0  2.8  200  396 p7 S    Aug 12  0:02 /usr/local/bin/zsh
mc          301  0.0  0.0   16    0 ?  IW   Aug 12  0:08 screenblank
mc          715  0.0  0.0   24    0 ?  IW   Aug 12  0:00 /usr/gnu/libexec/emacs/1

mc         9301  0.0  0.0  104    0 p9 IW   Aug 13  0:00 /usr/ucb/ftp -i -n -g -v

frobozz:~: screendump mgrscreen.raster
```

Guess what is the architecture?

My First Desktop at Work



CLI versus GUI

- Advantages of CLI
  - Faster and easier for experts (not beginners)
  - Scripting for automations
  - Easier to be run in batches, either by remote users or as cron jobs
  - More precise than mouse/GUI
- Disadvantages of CLI
  - Some applications (Photoshop?) are impossible
  - Learning curves are steep

WHAT IS UNIX

UNIX is ..

- A multi-user, multi-tasking operating system
- Machine independent system built on C (instead of Assembly)
- Software development environment
  - With plenty of utilities
- Created in 1969 at Bell Labs in Murray Hill, NJ

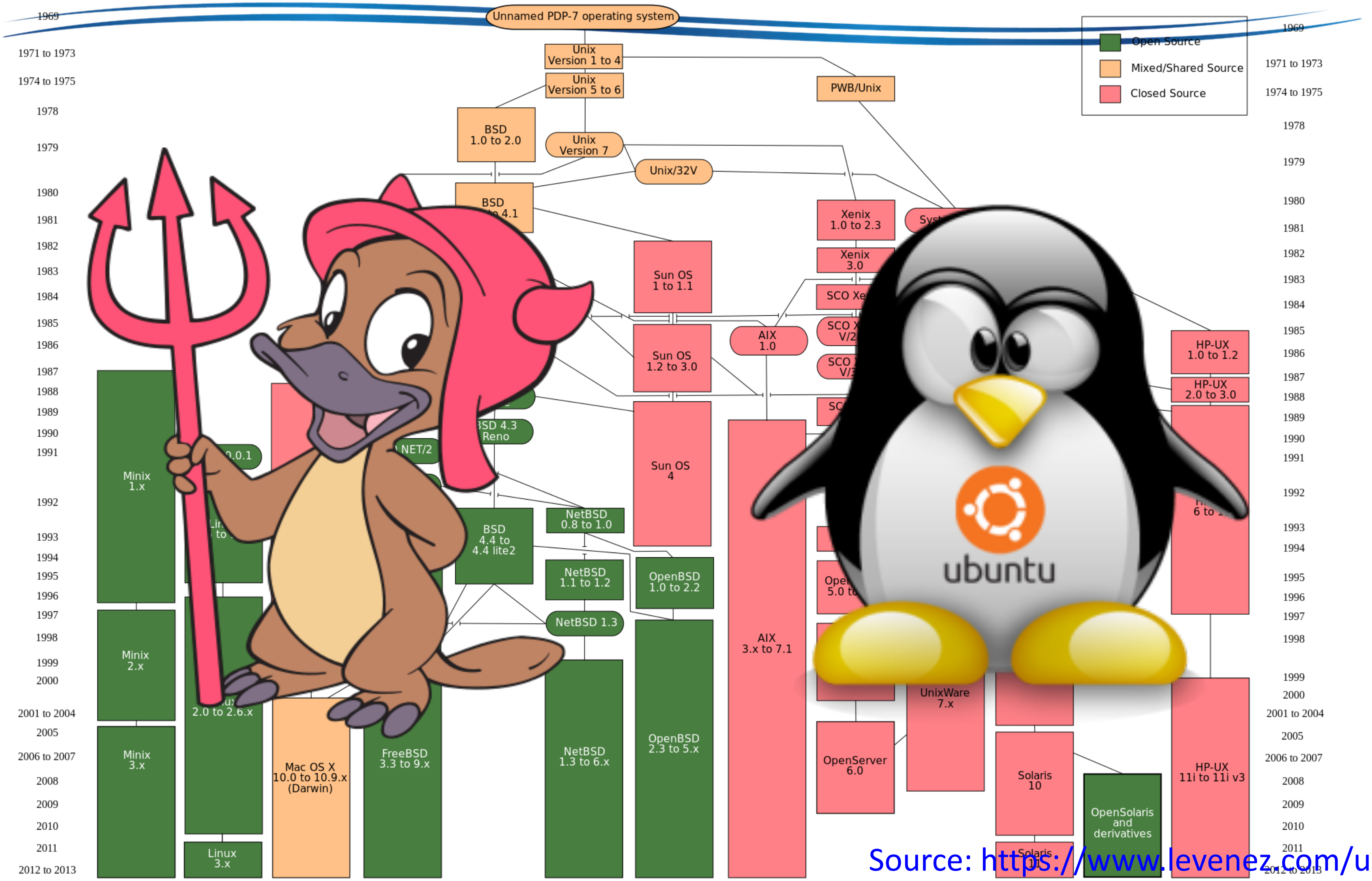
UNIX Philosophy

- Short names for utilities: e.g., `list` versus `ls`
- Each utility does one thing well, and one thing only
- Allow users to *pipe* each utility's output to another utility's input to accomplish complicated tasks
  - Count how many times my students mentioned my name in their theses
    - `%grep -i bear */*/*.tex */*/*/*.tex | wc -l`
    - `%find . -name '*.tex' -exec grep -i bear {} \;`
  - You are only limited by your imagination
- Get a utility working first, then make it better

Why UNIX

- Portability
- Productivity ← Scripts that automates the executions of tasks that can alternatively done by human one-by-one ← tedious!
- Multi-tasking and distributed processing ← TCP/IP

History of Unix-Like Systems



Source: https://www.levenez.com/unix/

Prepare Your Linux Environment

- Virtual machines: VMWare Player, VirtualBox, Parallels
  - Dual network interfaces: NAT and host-only
- Recommend Ubuntu (Debian) packages
  - gcc, g++, gdb, make
  - manpages-dev, manpages-posix, manpages-posix-dev
  - `#sudo apt-get install gcc g++ gdb make manpages-dev manpages-posix manpages-posix-dev`

ABOUT THE COURSE

Course Format

- Time: Mondays 3:30 p.m. - 5:20 p.m., Thursdays 2:20 - 3:10 p.m.
- Location: EECS 132
- Office hour: Thursdays 3:30 p.m.- 4:20 p.m., Delta 643
- TA: Chen-Nien Mao (gloiremao AT gmail.com), Delta 713
- Labs (weekly assignment demo): Tuesdays 7:00 - 9:00 p.m. at EECS 328.
- Website: http://nmsl.cs.nthu.edu.tw/index.php/courses

Prerequisites and Textbook

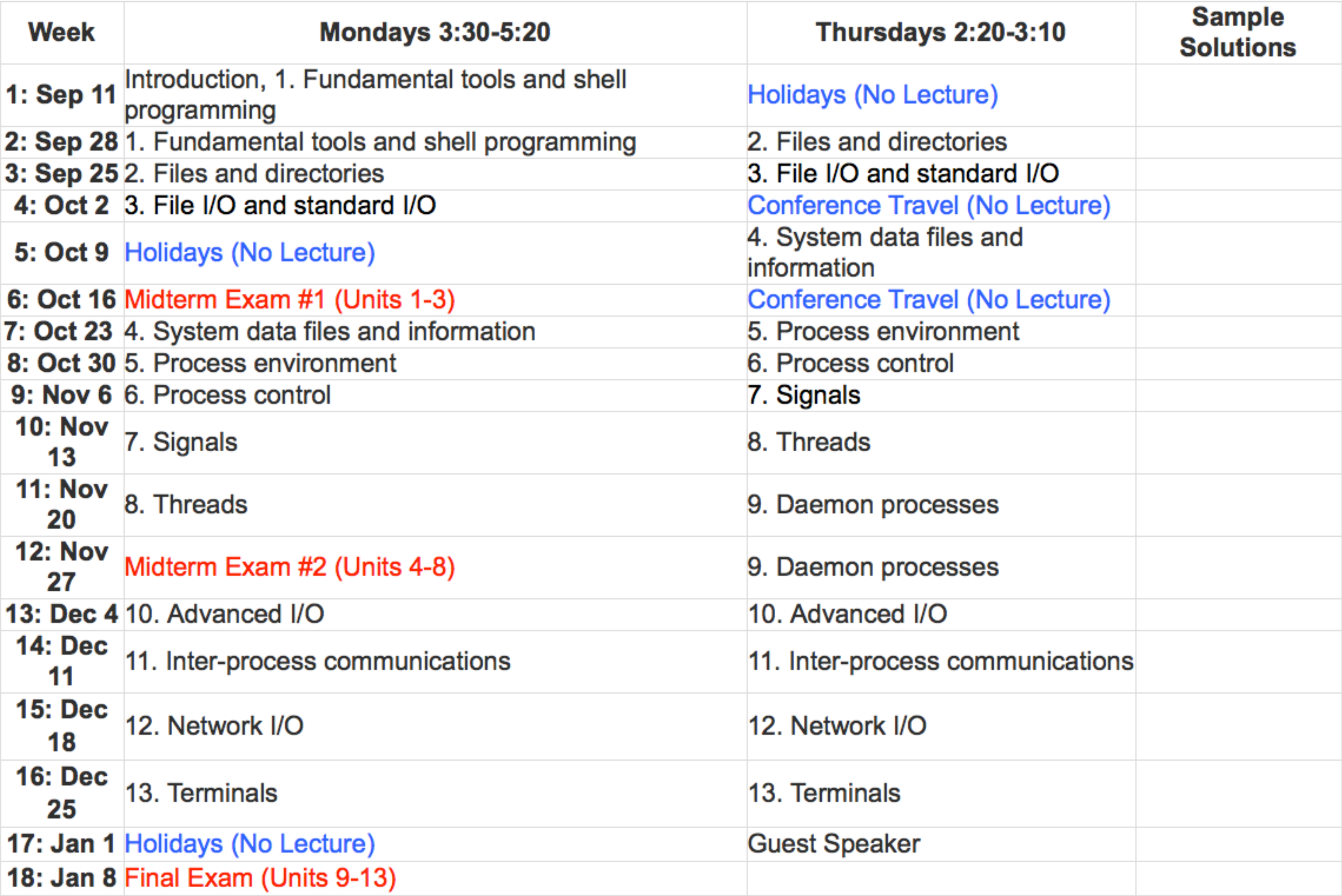
- Prerequisites:
  - Introduction to programming: must know C
  - Operating systems : heard of Inter-Process Communications (IPC)
- Textbook: W. Richard Stevens and Stephen A. Rago, "Advanced Programming in the UNIX Environment," 3rd ed., Addison Wesley
  - Safari version: http://proquest.safaribooksonline.com/book/programming/unix/9780321638014
  - Please read the book, even though I may not be able to cover everything in lectures

Advanced Programming in the UNIX Environment
Third Edition
W. Richard Stevens
Stephen A. Rago
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Grading Policy

- Weekly assignments (55% + 5% Bonus): 12 times, 5% each
  - Assignments are given on the last slide of each topic
  - Students turn in their assignments during weekly labs
  - The TA grades assignments during labs
  - Scores will be announced on iLMS
  - No make-up demos unless approved by the instructor **before** the lab session.
  - Late submissions within one week are subject to 50% penalty; submission beyond one week won't be graded.
- Midterms (30% Bonus): Two times, 15% each
- Final Exam (15%)
- No curving.....

Tentative Schedules



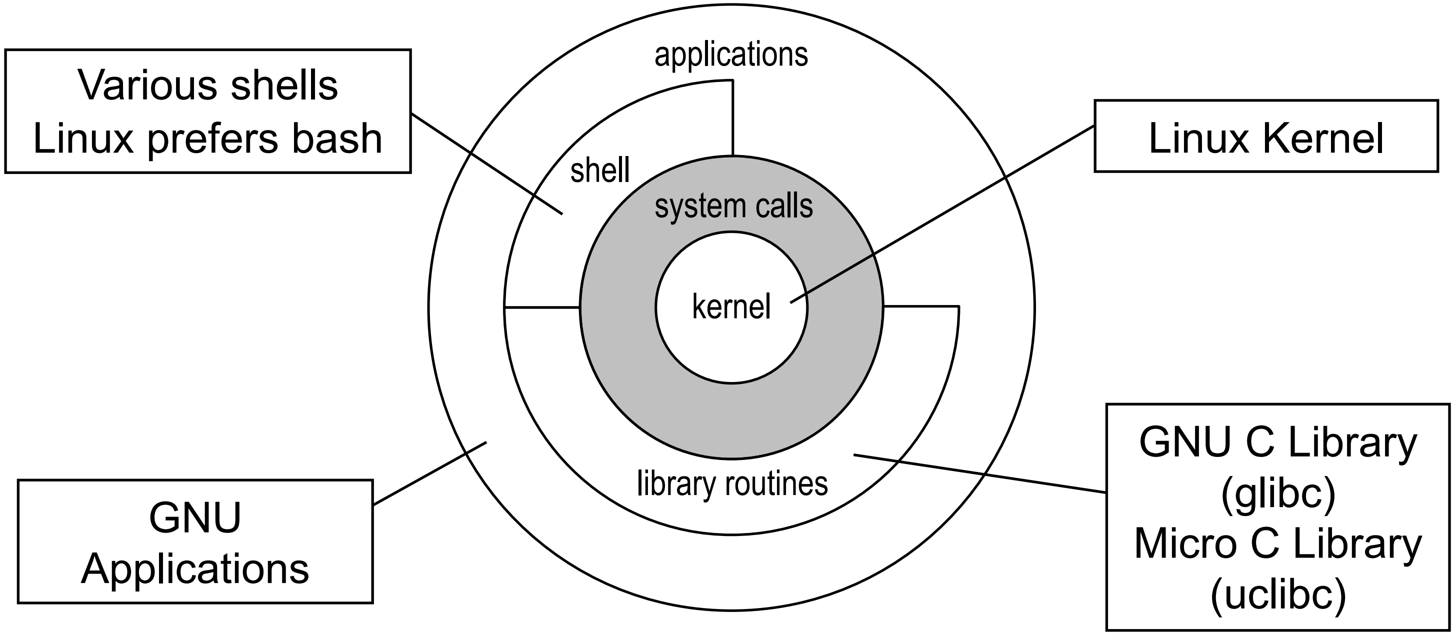
Week	Mondays 3:30-5:20	Thursdays 2:20-3:10	Sample Solutions
1: Sep 11	Introduction, 1. Fundamental tools and shell programming	Holidays (No Lecture)	
2: Sep 28	1. Fundamental tools and shell programming	2. Files and directories	
3: Sep 25	2. Files and directories	3. File I/O and standard I/O	
4: Oct 2	3. File I/O and standard I/O	Conference Travel (No Lecture)	
5: Oct 9	Holidays (No Lecture)	4. System data files and information	
6: Oct 16	Midterm Exam #1 (Units 1-3)	Conference Travel (No Lecture)	
7: Oct 23	4. System data files and information	5. Process environment	
8: Oct 30	5. Process environment	6. Process control	
9: Nov 6	6. Process control	7. Signals	
10: Nov 13	7. Signals	8. Threads	
11: Nov 20	8. Threads	9. Daemon processes	
12: Nov 27	Midterm Exam #2 (Units 4-8)	9. Daemon processes	
13: Dec 4	10. Advanced I/O	10. Advanced I/O	
14: Dec 11	11. Inter-process communications	11. Inter-process communications	
15: Dec 18	12. Network I/O	12. Network I/O	
16: Dec 25	13. Terminals	13. Terminals	
17: Jan 1	Holidays (No Lecture)	Guest Speaker	
18: Jan 8	Final Exam (Units 9-13)		

Questions So Far?

If not, let’s have a break

INTRODUCTION TO UNIX ENVIRONMENT

UNIX Architecture



Various shells
Linux prefers bash

Linux Kernel

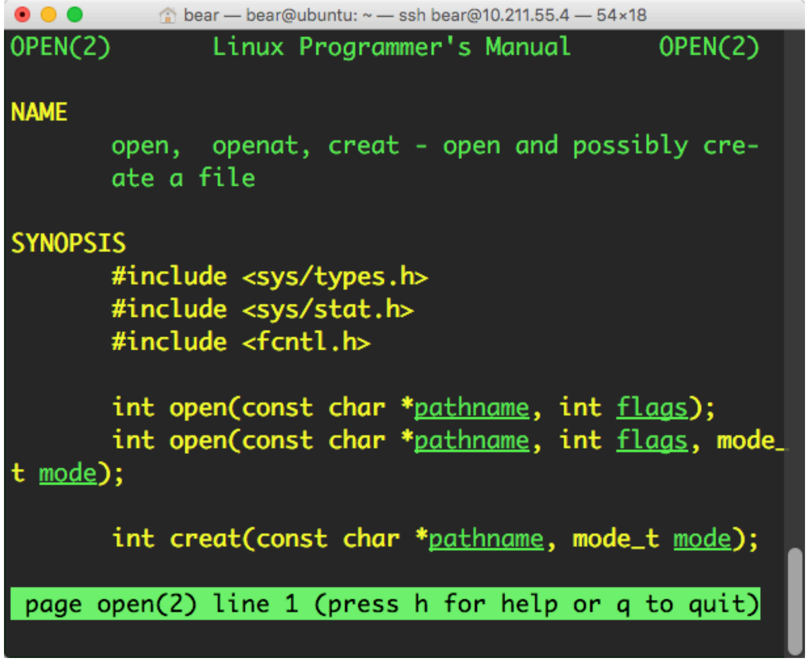
GNU Applications

GNU C Library (glibc)
Micro C Library (uclibc)

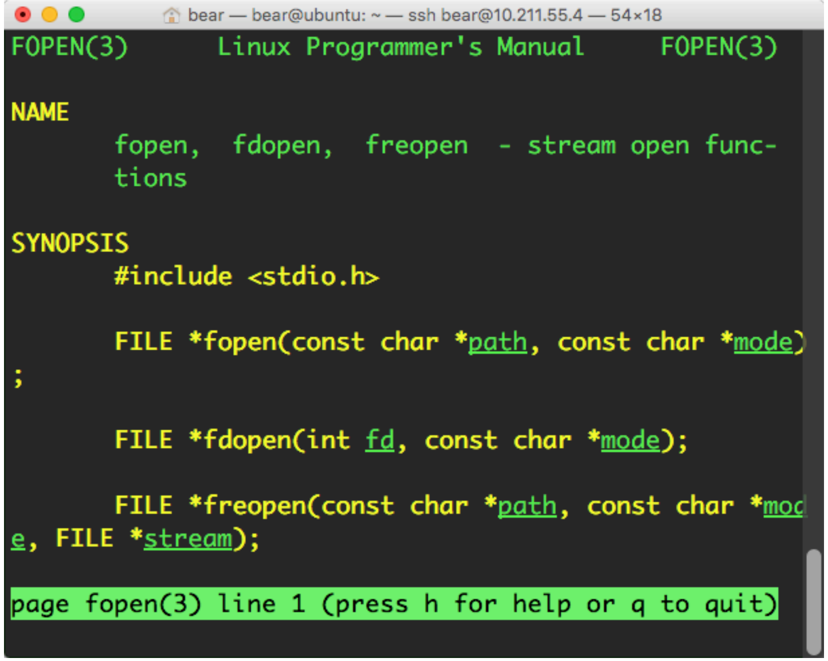
- Linux system calls
  - http://man7.org/linux/man-pages/man2/syscalls.2.html

System and Library Calls

- System calls: entry points into kernel-space code
- Library calls: shared user-space functions

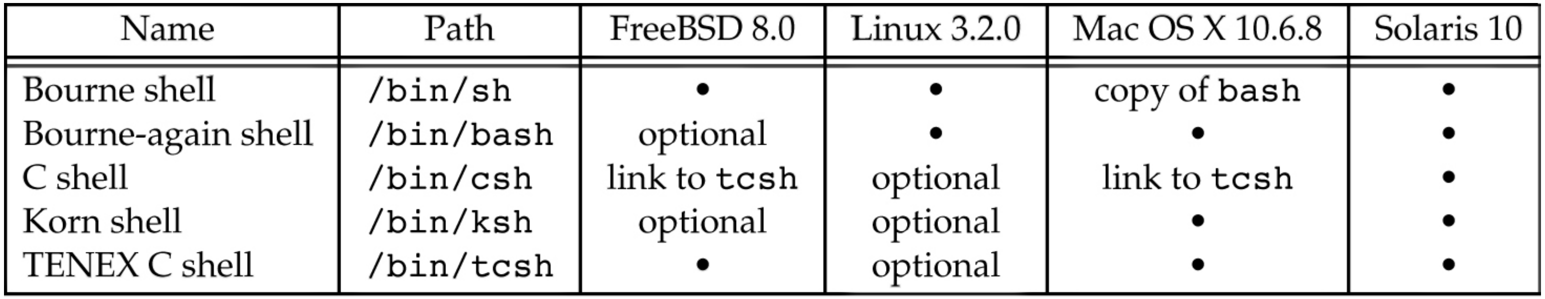


```
OPEN(2)        Linux Programmer's Manual       OPEN(2)

NAME
       open,  openat, creat - open and possibly cre-
       ate a file

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <fcntl.h>

       int open(const char *pathname, int flags);
       int open(const char *pathname, int flags, mode_
t mode);

       int creat(const char *pathname, mode_t mode);

 page open(2) line 1 (press h for help or q to quit)
```



```
FOPEN(3)       Linux Programmer's Manual      FOPEN(3)

NAME
       fopen,  fdopen,  freopen  - stream open func-
       tions

SYNOPSIS
       #include <stdio.h>

       FILE *fopen(const char *path, const char *mode)
;

       FILE *fdopen(int fd, const char *mode);

       FILE *freopen(const char *path, const char *mod
e, FILE *stream);

page fopen(3) line 1 (press h for help or q to quit)
```

Popular Shells



Name	Path	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
Bourne shell	`/bin/sh`	•	•	copy of `bash`	•
Bourne-again shell	`/bin/bash`	optional	•	•	•
C shell	`/bin/csh`	link to `tcsh`	optional	link to `tcsh`	•
Korn shell	`/bin/ksh`	optional	optional	•	•
TENEX C shell	`/bin/tcsh`	•	optional	•	•

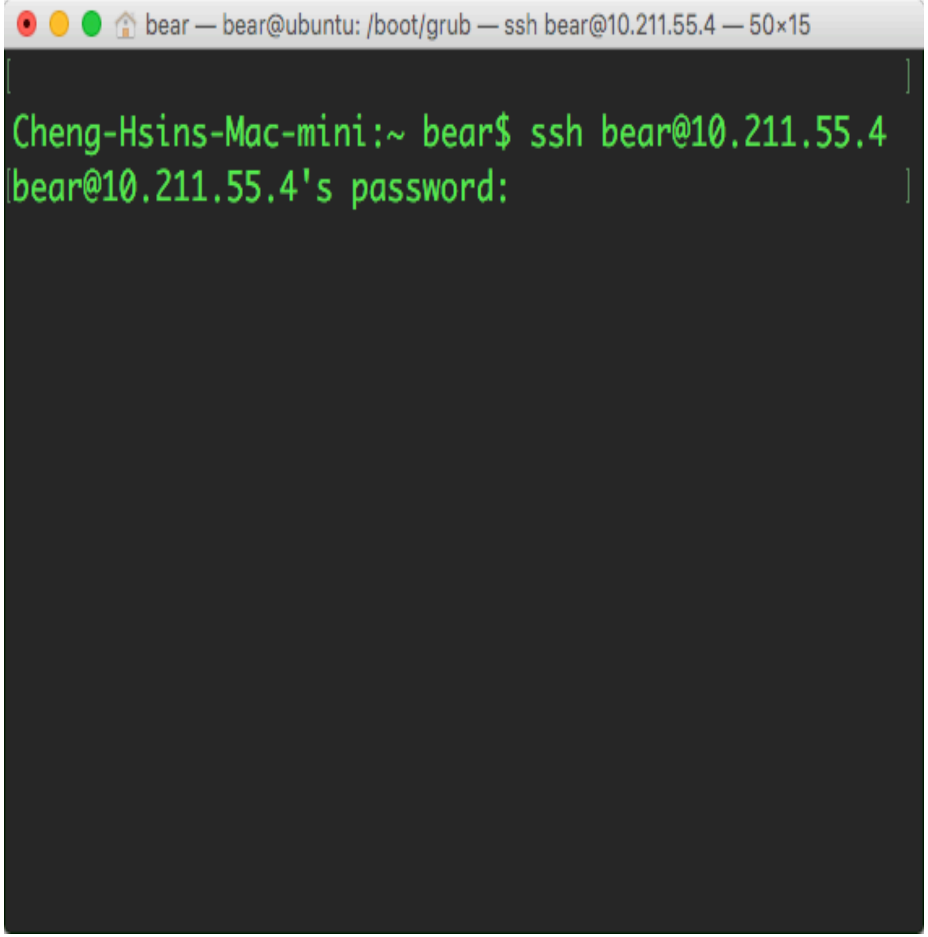
Figure 1.2 Common shells used on UNIX systems

Sample Difference	bash	tcsh
Variable	x=12 set x=12	set x=3
Environment Variable	export z=15	setenv z 15
PATH	export PATH=/a:/b	set path=(/a /b)
Startup File	~/.profile	~/.cshrc

Booting Process

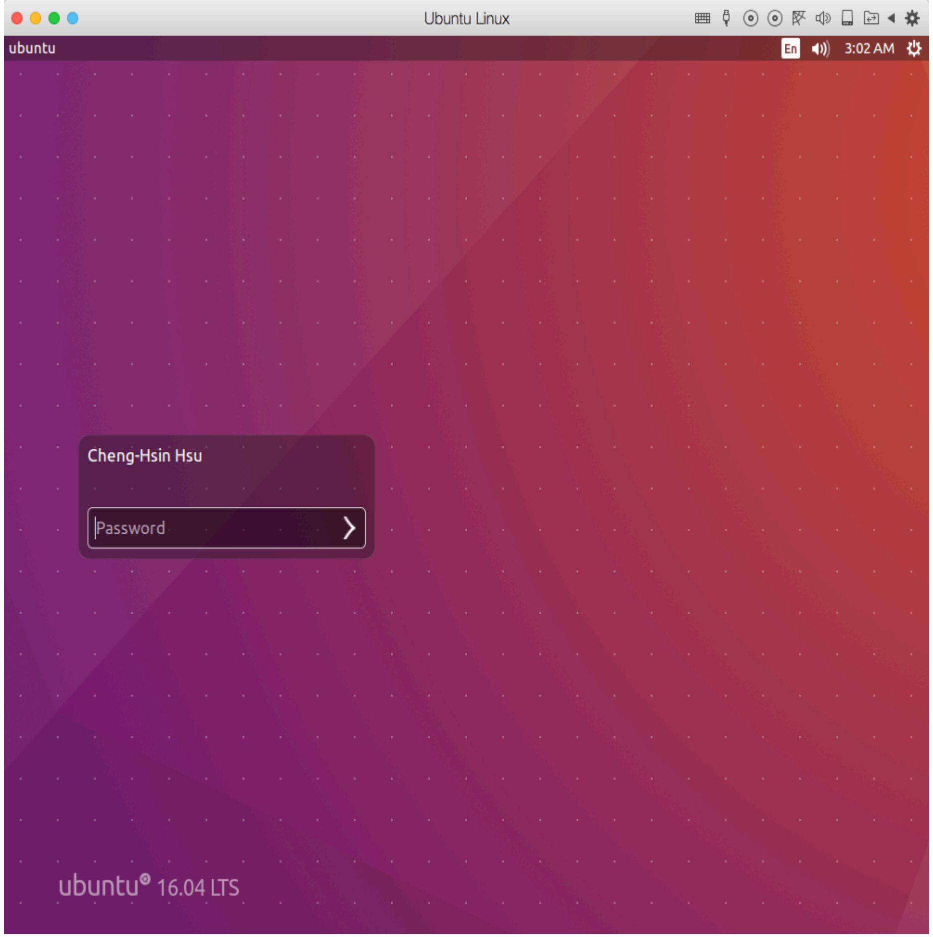
- OS loader (e.g., grub) loads a kernel and an (optional) RAM disk image into the memory
- Kernel initializes hardware components
- Kernel launches the first process
  - `/sbin/init, /etc/init, /bin/init, and then try /bin/sh` ←until one of them works
- The init process brings up the rest of everything
  - Mount file systems
  - Set up networks
  - Launch services
  - Provide the login prompt

First Impression



bear — bear@ubuntu: /boot/grub — ssh bear@10.211.55.4 — 50×15

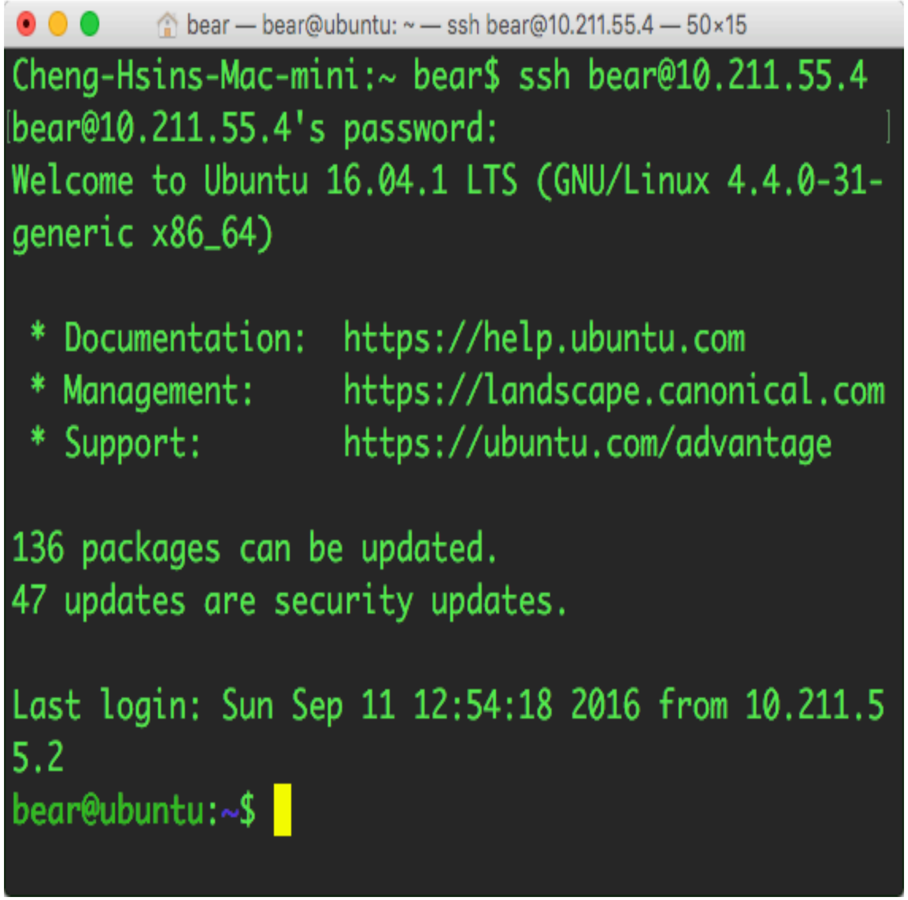
```
Cheng-Hsins-Mac-mini:~ bear$ ssh bear@10.211.55.4
bear@10.211.55.4's password:
```



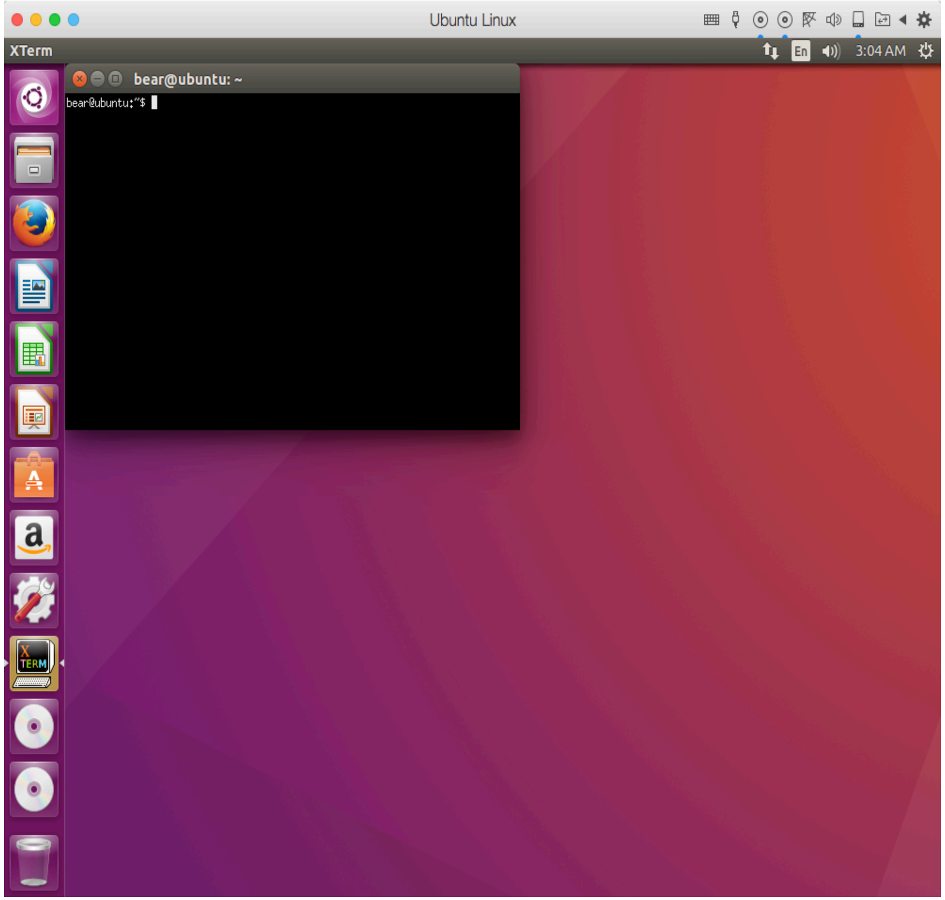
Cheng-Hsin Hsu

Password

Second Impression



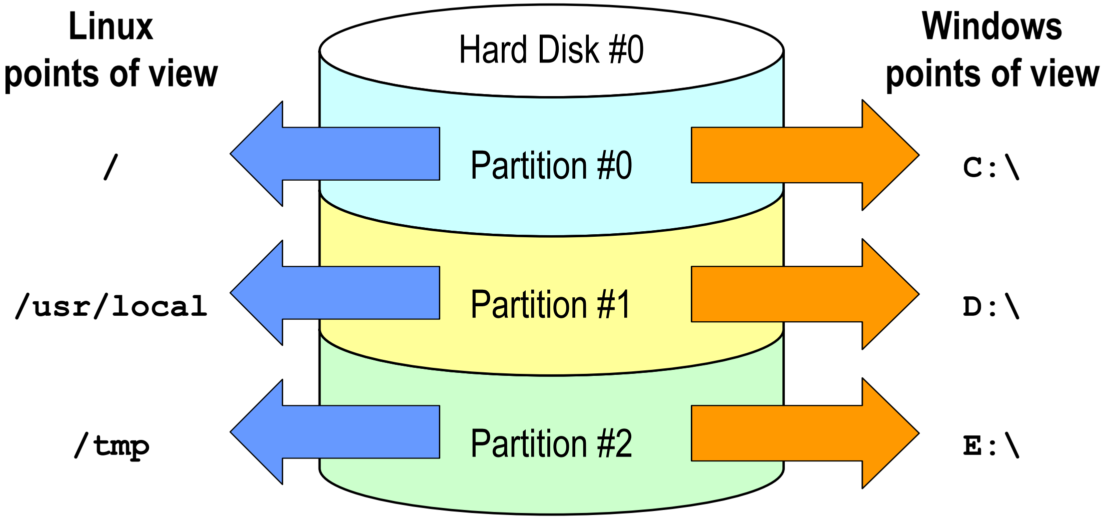
```
Cheng-Hsins-Mac-mini:~ bear$ ssh bear@10.211.55.4
bear@10.211.55.4's password:
Welcome to Ubuntu 16.04.1 LTS (GNU/Linux 4.4.0-31-
generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

136 packages can be updated.
47 updates are security updates.

Last login: Sun Sep 11 12:54:18 2016 from 10.211.5
5.2
bear@ubuntu:~$ 
```



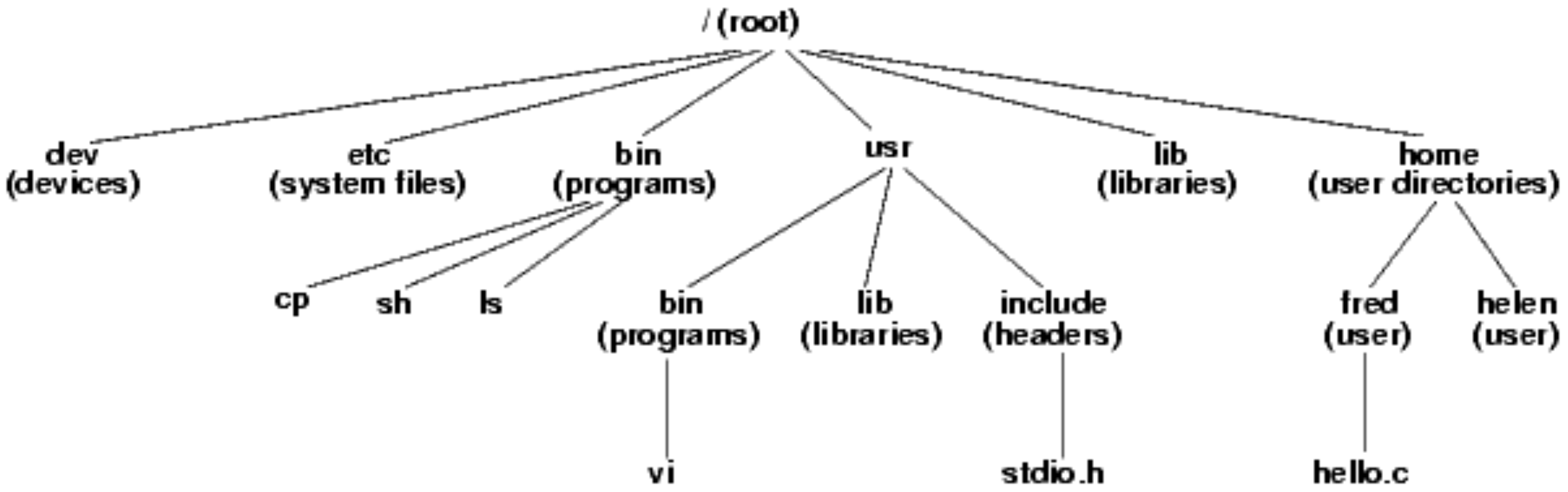
```
bear@ubuntu:~$ 
```

File System Architecture

- Hierarchical arrangement of directories and files
- Everything starts from the "root directory" (/)
- The "mount" program ← try it, also df
- Filenames (and commands) are case-sensitive
  - Hierarchical File System (HFS) on OSX by default is case insensitive ← an exception



Common Directory Structure



Basic UNIX Commands

- `ls`: list files
- `mkdir`: make directory
- `cd`: change directory
- `pwd`: print working directory
- `rmdir`: remove directory
- `cp`: copy file/directory
- `mv`: move
- `rm`: remove
- `cat`: concatenate and print

- `less` (or `more`): page splitter
- `echo`: print a string
- `date`: print or set the date and time
- `env`: print out all environment variables
- `touch`: change file time
- `tar`: archive tool

More UNIX Commands

- There are many UNIX commands, and it is impossible to cover all of them
- Built-in commands: Provided by login shells
- Other commands: Binaries installed by the system administrators
    - Often placed in standard locations: /bin, /sbin, /usr/bin, /usr/sbin, /usr/local/bin, /usr/local/sbin
    - Binaries are searched according the directories listed in the `PATH` environment
- Linux standard base (LSB)
    - What tools and libraries are mandatory for a Linux operating system
    - https://en.wikipedia.org/wiki/Linux_Standard_Base
- (Linux) Filesystem Hierarchy Standard (FHS), is one part of the LSB
    - Recommended locations to placed your files
    - https://en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard

Conventions in Documents

- Run command as a regular user
  - % *command* … (sometimes we use $ instead of %)
- Run command as a privileged user (super user, or root)
  - # command … (don't be confused with comments)
- Related to command arguments
  - Square brackets [ ]: optional part, e.g., cat [filename]
  - Dots …: multiple arguments are allowed, e.g., cat [file …]
  - Dash - or --: options for a command, e.g.,
    - Part of options for the ls command: -a, --all, --color, -F, …
    - Single dash options may be aggregated: -aF

Redirection and Pipe

- Redirection
  - Outputs of a command can be stored in a file
    - % echo Hello, World! > a.txt
    - % echo Hello, World! >> a.txt
  - File content can be used as inputs to a command
    - % cat < a.txt
- Pipe
  - Outputs of a command can be inputs of another command
    - % echo Hello, World! | cat
    - % cat hello.c | less
  - Pipe can be chained
    - % echo Hello, World! | tr a-z A-Z | cat

Manual Pages are Your Friends

- The man(1) command
  - The command you must know in the UNIX world!
- Manual pages for commands, system calls, library functions, kernel routines, …
- Basic usage
  - `$ man [section] page` ← `man 2 open`
  - `$ man -k regexp` ← `man –k prints`
- Convention – `page(section)`
  - Examples: `ls(1),man(1), read(2), crypt(3),tty(4),shadow(5), printf(1),printf(3),`…

Man(ual) Page Sections

1. Executable programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within program libraries)
4. Special files (usually found in /dev)
5. File formats and conventions eg /etc/passwd
6. Games
7. Miscellaneous (including macros and conventions)
8. System administration commands (only for root)
9. Kernel routines [Non standard]

FUNDAMENTAL UNIX PROGRAMMING PRACTICES

"Hello World”

```
#include <stdio.h>

int main() {
        printf("Hello, World.\n");
        return 0;
}
```

- Compile and run "Hello, World." in the UNIX environment
  - `$ gcc hello.c     // this generates a.out`
  - `$ ./a.out`
  - `$ gcc hello.c -o hello  // this generates hello`
  - `$ ./hello`

Return from the main() Function

- Return value of the main() function
  - It is actually a one byte value
  - Return zero: the value indicates 'True', or no problem
  - Return non-zero values: the values indicate 'False', or error
  - Can be used to determine program execution status
- Read return values from your program
  - Run '`echo $?`' immediately right after your program execution
- Try it yourself….
  - return -999;
  - echo $? ← what do you get?
  - WHY?

More about Return Values

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    return atoi(argv[1]);
}
```

- Compile it with `gcc test.c -o return`
- What does this program do?
- What’s the problem with this code?

Boolean OR and AND

- Shell’s short-cut branch
  - Break an evaluation when the final result is known
- Boolean OR (||) – Stop evaluation when a condition is true
  - `$ ./return 0 || echo 'A'`
  - `$ ./return 1 || echo 'B'`
- Boolean AND (&&) – Stop evaluation when a condition is false
  - `$ ./return 0 && echo 'C'`
  - `$ ./return 1 && echo 'D’`
- How about `$ ./return 0 | echo 'A'`

Arguments

```
int main(int argc, char *argv[]) {
    int i;
    for(i = 0; i < argc; i++)
        printf("'%s' ", argv[i]);
    printf("\n");
    return 0;
}
```

- What will be the outputs?
  - $ ./args
  - $ ./args a b c d
  - $ ./args "a b c d"
  - $ ./args 'a b c d'
  - $ ./args "home = $HOME”
  - $ ./args 'home = $HOME’
  - $ ./args 12

Handle Options

- The getopt(3) and getopt_long(3) style options
- Used by many UNIX utilities
- getopt(3) reads dash plus single character options (short options)
  - Options can be aggregated
  - For example, -a -b is equivalent to -ab
- getopt_long(3) also reads double-dash plus key word options (long options)
  - For example, --all, --color

Getopt(3)

```
int getopt(int argc, char * const argv[], const char *optstring);
```

- argc: the argc parameter received by the main function
- argv: the argv parameter received by the main function
- optstring: list of valid option characters (usually colons and alphabets)
- Add a colon (:) right after an option character indicates that the option requires an additional argument
- Common return value of getopt(3)
  - -1: No more options
  - Colon (:) or question mark (?): Invalid option encountered
- Global variables
  - optind: An integer stores the number of arguments consumed by getopt(3)
  - optarg: A string points to the additional argument of the current option (if : is given)

UNIX Time Representations

- Wall clock time: `time_t`:
  - Number of seconds elapsed from 00:00:00, January 1st, 1970 UTC (the "epoch")
  - It is often a 32-bit signed integer
  - Will be overflowed after 03:14:07 January 19th, 2038 – The year 2038 problem!

- High precision time: `struct timeval`, in microsecond unit
  - Basically `time_t` plus a microsecond precision timestamp

- CPU time: `clock_t`, in CPU-ticks unit
  - CLOCKS_PER_SEC constant
  - POSIX requires CLOCKS_PER_SEC to be 1,000,000 independent of the actual clock resolution

From `select(2)`:

```
struct timeval {
    long tv_sec;   // s
    long tv_usec;  // ms
};
```

UNIX Time Representations (cont.)

- time(3) function: Get time in time_t format

```
time_t time(time_t *t);
```

- gettimeofday(2) function: Get time in struct timeval format

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

  - tz is obsoleted, it should be NULL

- clock(3) function: Get time in clock_t format

```
clock_t clock(void);
```

Measure Program Performance

- A simple metric: Program running time
- A simple example: the time command

```
$ time sleep 10
real    0m10.003s
user    0m0.000s
sys     0m0.003s
```

- Real time, user time, and sys time
- How to get these numbers in programs?
  - gettimeofday(3): get wall clock time in microsecond precision (in timeval format)
  - clock(3): get CPU ticks (user + sys)
  - getrusage(3): get CPU time (in timeval format)

Measure Program Performance (cont.)

```
t0 = get_the_current_timestamp();

// The codes we want to measure ...
Do something ...

t1 = get_the_current_timestamp();

Compute and output (t1 – t0)
```

Error Handling

- Check function return values
  - (Integer) Zero or positive values: return without errors
  - (Integer) Negative values (usually -1): return with errors
  - (Pointer) Non-NULL: return without errors
  - (Pointer) NULL: return with errors
  - This is applicable for most of the C library functions
- What kinds of error?
  - Determine using the `errno` variable.
  - A global variable built in C library
    - It is not thread-safe!
    - But not a problem if your system supports Thread Local Storage (TLS)
  - Check it right after receiving an error return value
- List of error codes
  - See `errno(3)` manual pages

Display Errors

- Required headers and declarations

```
#include <stdio.h>   // for perror
#include <string.h>  // for strerror
#include <errno.h>   // errno variable, and defs of error codes
```

- Convert an error number to a human-readable string
  - `strerror`
  - `perror`

```
Printf("error = %s\n", strerror(errno));
perror("some prefix");
```

Error Recovery

- Fatal errors
  - No way to recovery
  - Show error messages, log, and then exit
- Non-fatal errors
  - May be temporary errors ← can be handled….
  - Delay for a short time and then retry
  - Examples
    - EAGAIN, ENFILE, ENOBUFS, EWOULDBLOCK, …

GNU TOOLCHAIN

The Compiler

- gcc – GNU C Compiler
- g++ – GNU C++ Compiler
- Frequently used options
  - `-S`: do not compile, generate assembly only (output to .s)
  - `-E`: do not compile, perform preprocessing only (output to stdout)
  - `-c`: compile only, do not link
  - `-g`: embed debugging information
  - `-Wall`: turn on all warnings
  - `-l`: link with a library, e.g., `-lxxx` will link with a library named `libxxx.a`
  - `-I`: add include path, e.g., `-I/usr/local/include`
  - `-L`: add library path, e.g., `-L/usr/local/lib`

Compile a Single Source Code

- Compile and generate the executable binary
  - % g++ hello.cpp (the output will be a.out)
  - % g++ hello.cpp -o hello (the output will be hello)
- Execute the executable
  - % ./a.out (or ./hello)
  - Do not miss the ./ prefix
- Try -E and -S options

Compile Multiple Source Code Files

- Suppose you have s1.cpp, s2.cpp, and s3.cpp
- Strategy #1
  - g++ s1.cpp s2.cpp s3.cpp -o output (generates output)
- Strategy #2
  - g++ -c s1.cpp (generates s1.o)
  - g++ -c s2.cpp (generates s2.o)
  - g++ -c s3.cpp (generates s3.o)
  - g++ s1.o s2.o s3.o -o output (generates output)
- Which one is better?

Linking C and C++ Files (1/3)

- Suppose we have two source code files, a.c and b.cpp

```
a.c:

int b();
int main() {
    return b();
}
```

```
b.cpp:

int b() {
    return -1;
}
```

- We then compile and link the two files:
  - gcc -c a.c (generates a.o)
  - g++ -c b.cpp (generates b.o)
  - g++ -o test a.o b.o (does it work?)

Linking C and C++ Files (2/3)

- Let’s check what we have in the object codes
- We can use the nm tool to dump symbols

```
$ nm a.o
                 U b
0000000000000000 T main
```

```
$ nm b.o
0000000000000000 T _Z1bv
```

- Name mangling
- How to solve this?

Linking C and C++ Files (3/3)

- We have to modify b.cpp if a function will be called from a C program

```
b.cpp:

#ifdef __cplusplus   (only needs for a C++ compiler)
extern "C" {    (declare that everything within the scope)
int b();        (should be treated as C symbols, not C++ )
}
#endif

int b(int n) {
    return n;
}

int b() {
    return b(-1);
}
```

```
$ nm b.o
000000000000000c T b
0000000000000000 T _Z1bi
```

MAKE AND MAKEFILE

Why Make and Makefile

- Project management
  - Simplify build processes
  - Manage project dependencies
- A common scenario
  - Build a program with multiple source files
- Steps
  - Write rules in a file named *Makefile*
  - Run the *make* command
    - By default, make run *the first* rule in the makefile

The make Command

- Simply type ‘make’ in the command prompt
  - `$ make`
  - Or alternatively, specify a target rule:
    - `$ make clean`
- Common options
  - `-C {dir}`: switch to the given directory and run the make command
  - `-f {makefile}`: specify a different filename
  - `-I {dir}`: specify include directory search path
  - `-j {n}`: allow simultaneously jobs (commands)
  - For the details, see the man page!

The Makefile

- Rule definitions
- Variable definitions
- Automatic variables
- Special rules
- Pattern rules

Rule Definitions

- General format
  - rulename: dependencies (or prerequisites)
    (tab) rules
- Rulename – the target to be built
- Dependencies
  - Prerequisites required to build the target
  - Separated by spaces
- Rules
  - Commands to build the target

Rule Definitions (Cont’d)

- Comments: start with a pond sign (#)
- Split a single line into multiple lines: back slash (\)

An Example

```
test: test1.c test2.c     # comment
   gcc -c test1.c
   gcc -c test2.c
   gcc -o test test1.o test2.o
```

- This is *not* a good example
- Both the test1.c and the test2.c files are compiled if either one of them is modified – we will refine it later …

Variable definitions

- Common usage
  - Set: VARNAME=value
  - Use: $(VARNAME)
- Create variables
  - `CC  = gcc`
  - `CXX = g++`
  - `CFLAGS = -I. -Wall`
- Use the variables
  - `$(CC) -c test.c $(CFLAGS)`
  - `A nonexistence variable == an empty string`

Automatic Variables

- $@: The target file name
- $<: The name of the first prerequisite
- $?: The name of all prerequisites that are *newer* than the target
- $^: The name of all prerequisites. *Duplicated entries will be removed*
- $+: Like $^, but duplicated entries will *not* be removed

A Refined Example

```
GCC = gcc
CFLAGS = -g –Wall

.c.o:      # old-fashioned!
   $(GCC) –c $< $(CFLAGS)

test: test1.o test2.o
   $(GCC) -o test $^
```

- This one is better!
- Only modified objects will be re-built

Special Rules

- .SUFFIXES (old fashioned!)
  - Add non-default suffixes (filename extensions)
  - Example

```
.SUFFIXES:              # (remove all)
.SUFFIXES: .asm .inc
```

- .PHONY
  - Targets are not files!
  - Example `.PHONY: all clean`

Pattern Rules – The % Symbol

- The filename between the prefix and the suffix are called "stem"
- Remember the old-fashioned ".c.o:" rule?
- It is equivalent to

```
%.o: %.c
```

- The new style provides much more flexibilities

Reference and Example

- A complete reference to make and Makefile
  – The make manual page
  – http://www.gnu.org/software/make/manual/

```
IDIR =../include
CC=gcc
CFLAGS=-I$(IDIR)

ODIR=obj
LDIR =../lib

LIBS=-lm

_DEPS = hellomake.h
DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))
```

```
_OBJ = hellomake.o hellofunc.o
OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ))

$(ODIR)/%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)

hellomake: $(OBJ)
	gcc -o $@ $^ $(CFLAGS) $(LIBS)

.PHONY: clean

clean:
	rm -f $(ODIR)/*.o *~ core
$(INCDIR)/*~
```

DEBUG WITH GDB

GDB – Quick Introduction

- A command line based (interactive) debugger
- All source codes must be compiled with -g!
  - Don’t strip the symbols
- Example #1
  - $ gcc -g test.c
- Example #2
  - Makefile
  - See CFLAGS

```
GCC = gcc
CFLAGS = -g -Wall

.c.o:      # old-fashioned!
    $(GCC) -c $< $(CFLAGS)

test: test1.o test2.o
    $(GCC) -o test $^
```

The First Impression

- $ gdb a.out *# a.out is the program executable*

```
GNU gdb (GDB) 7.10.1
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-unknown-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from a.out...done.
(gdb) _
```

Compiled without -g Option

- $ gdb a.out *# a.out is the program executable*

```
GNU gdb (GDB) 7.10.1
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-unknown-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from a.out...(no debugging symbols found)...done.
(gdb) _
```

Basic Commands

- Show source codes
  - list [line # | function | file:line # | file:function]
- Start to debug a program
  - run [arguments …]
- Run the next command
  - next (will *not* enter a function)
  - step (will enter a function)
- Display
  - print

Breakpoints

- Set breakpoints
  - break [line # | function | file:line # | file:function]
- Delete breakpoints
  - clear [line # | function | file:line # | file:function]
- Show breakpoints
  - info breakpoints
- Run until a breakpoint is reached
  - continue

Sample Source Code

- Source code: debug.c

```c
#include <stdio.h>

int main() {
    int i;
    char hello[] = "Hello, World!\n";
    char *ph = hello;
    for(i = 0; ph[i]!='\0'; i++) {
        putchar(ph[i]);
     }
    return 0;
}
```

Compile, Load, and Run

```
$ gcc -g debug.c -o debug
$ gdb debug
(gdb) run
Starting program: /tmp
Hello, World!

Program exited normally.
(gdb)
```

List Source Codes

```
(gdb) list 1
1       #include <stdio.h>
2
3       int
4       main() {
5               int i;
6               char hello[] = "Hello, World!\n";
7               char *ph = hello;
8               for(i = 0; ph[i]!='\0'; i++) {
9                       putchar(ph[i]);
10              }
(gdb)
```

Set Breakpoints and Run

```
(gdb) b 8
Breakpoint 1 at 0x8048485: file bug.c, line 8.
(gdb) run
Starting program: /raid/home/chuang/tmp/bug

Breakpoint 1, main () at hello.c:8
8               for(i = 0; ph[i]!='\0'; i++) {
(gdb) n
9                       putchar(ph[i]);
(gdb) print i
$1 = 0          # print a variable
(gdb) print ph[i]
$2 = 72 'H‘     # print another variable
(gdb) print putchar(ph[i])
$3 = 72         # run a function, may also consider 'call'
(gdb) print printf("%c\n", ph[i])
HH          # note that we are using bufferred I/O
$4 = 2
(gdb)
```

Debug a Crashed Program

- A buggy program …

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void walk(int depth) {
  int *p = 0;
  printf("%d\n", depth);
  if(rand()%5==0) *p = depth;
  else walk(depth+1);
  return;
}

int main() {
  srand(time(0));
  walk(0);
  return 0;
}
```

Debug a Crashed Program

- Run the program

```
$ gcc -g bug.c -o bug
$ ./bug
0
1
2
3
4
Segmentation fault (core dumped)
$ bear@ubuntu:/tmp$ ls -l bug core
-rwxrwxr-x 1 bear bear   9952 Sep 11 22:10 bug
-rw------- 1 bear bear 393216 Sep 11 22:16 core
```

Debug a Crashed Program

- Load the core and show call stack

```
$ gdb bug core
...
Reading symbols from bug...done.
[New LWP 5920]
Core was generated by `./bug'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x000000000040064e in walk (depth=3) at bug.c:8
8       if(rand()%5==0) *p = depth;
(gdb) bt
#0  0x000000000040064e in walk (depth=3) at bug.c:8
#1  0x000000000040065f in walk (depth=2) at bug.c:9
#2  0x000000000040065f in walk (depth=1) at bug.c:9
#3  0x000000000040065f in walk (depth=0) at bug.c:9
#4  0x0000000000400681 in main () at bug.c:15
```

No Coredump File?

- You may have to modify your system configuration using ulimit
  - ulimit -c unlimited
- Or change it permanently if you are a sudoer
  - Take Ubuntu Linux as an example: /etc/security/limits.conf

```
# /etc/security/limits.conf
#
#Each line describes a limit for a user in the form:
#
#<domain>        <type>  <item>  <value>
#
...
#*               soft    core            0
*                soft    core            1000000
#root            hard    core            100000
#*               hard    rss             10000
...
```

QUESTION?

Assignment #1 (5% Bonus)

- Implement your own light-weight wc utility, called lwc.c, in C (not C++)
  - (1%) lwc only supports three options -l, -w, and –c; lwc assumes at least one option is provided; lwc only process files (ignore stdin)
  - (3%) lwc supports multiple options; lwc ignore the order of options. The no. lines is always printed first, followed by the no. words and characters. ← run wc on Ubuntu to make sure that your outputs are identical to it!
  - (1%) If an invalid option or filename is given, lwc prints the same error message wc would print to stderr, and return the same non-zero exit status
- Submit your lwc.c in ILMS. Your code must be compiled with zero warning and error on our Ubuntu 16.04 LTS, to get any points
- Due date: September 20th