System Programming

Module 0

EGCI 252
System Programming

Computer Engineering Department
Faculty of Engineering
Mahidol University

What is System Programming?

In the past, **System Programming** was simple and well defined. It **involved building new services into the operating systems** (e.g. new device drivers). The activity often required specialized knowledge of the underlying operating system and employed services not normally used by the applications programmer.

What is System Programming? (cont.)

Nowadays, there are relatively few opportunities to contribute code to an operating system since for the core services it is a well established and a mature product. Possibly all the fundamental utilities and services have already been provided or will be provided by the vendor, therefore building new services into the kernel is not as common as it once was.

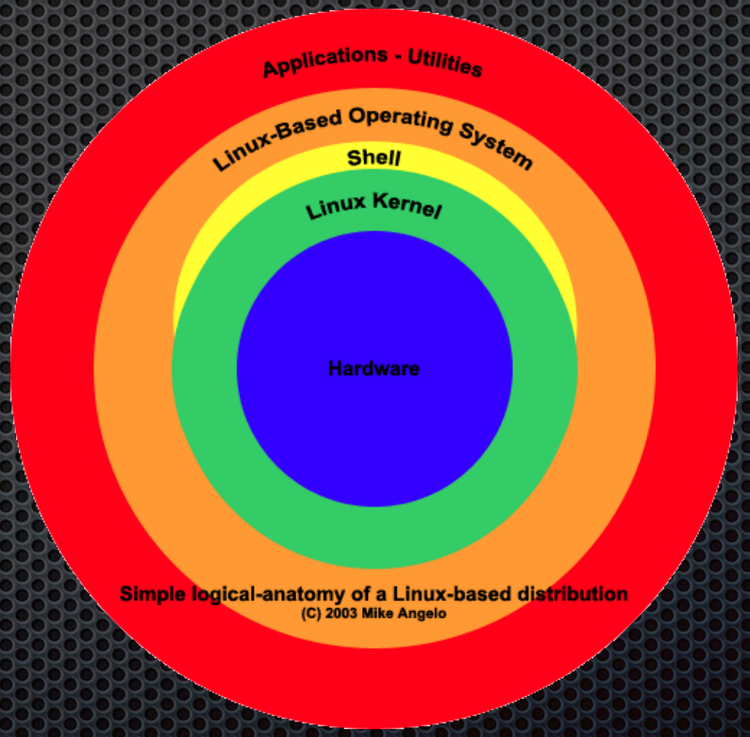
What is System Programming? (cont.)

The services and concepts of System Programming are likely to be an element of an application program which needs to make efficient use of the available resources, i.e. multiple process creation, synchronization of programs etc.. The distinction between system and application programming under LINUX/UNIX is less pronounced than other operating-systems, in fact some application programs eventually become part of the local system.

* Source: http://www.ee.ic.ac.uk/docs/Software/unix/programming/sys/intro.html

What is the LINUX kernel?

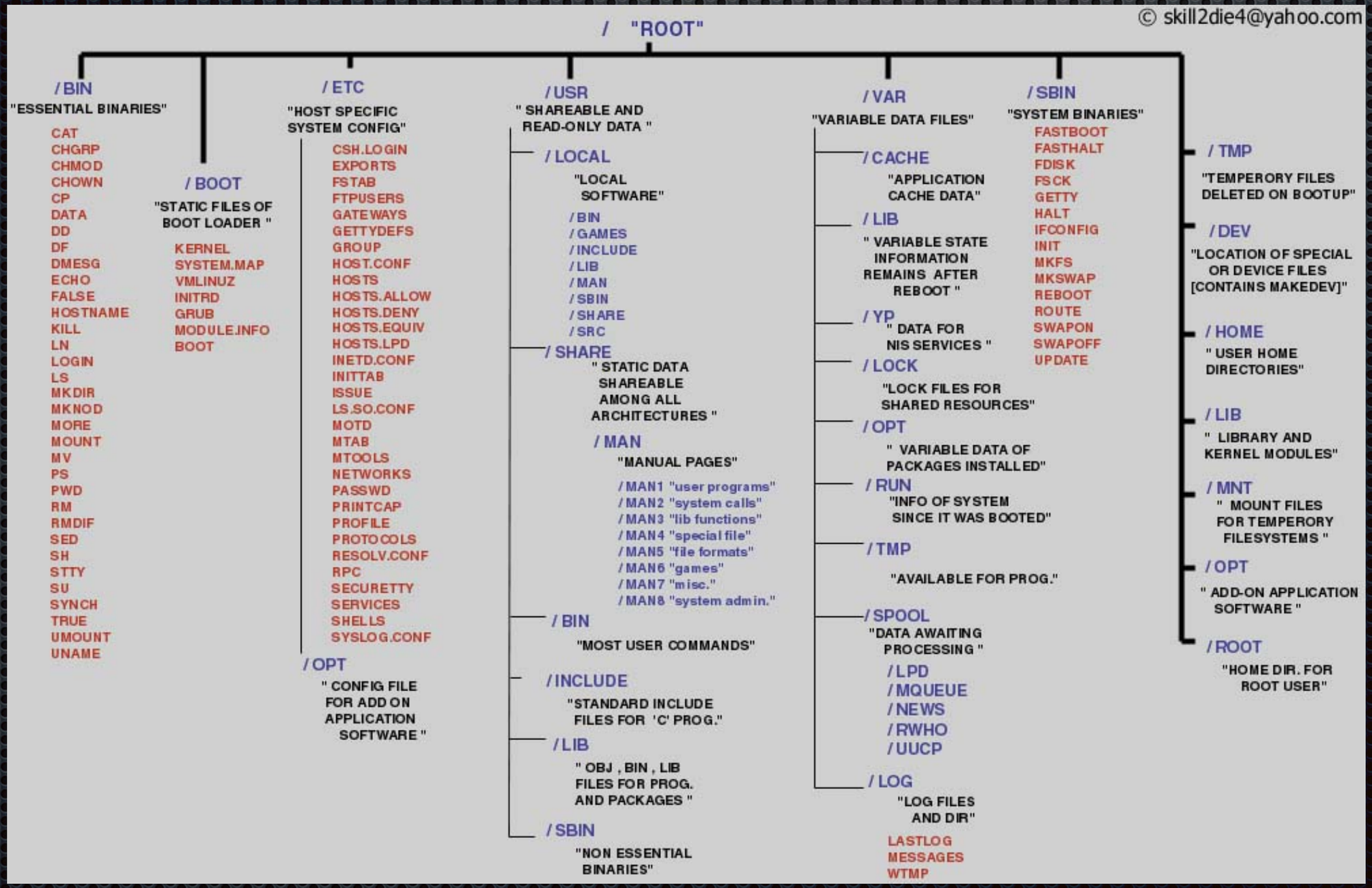
The innermost layer is the hardware which is wrapped in a program called the kernel which provides the services needed by the next layer, where application and utility software resides, i.e. the shells, utilities and user developed software.



UNIX/LINUX File System

- In UNIX/LINUX, everything is a file.
- A file has a name and administrative information called inode, which contains the length of the file and where on the disc it's stored.
- Programs can use serial ports, printers and other devices in the same way as they would use a file (e.g., open, close, read, write and ioctl)
- Directories are also files that hold the inodes and names of other files. (using an ls utility "ls –i" to show the list of file names and the inodes)

Diagram of UNIX/LINUX File System



Basic Unix/Linux Commands

* **pwd** (**p**rint the **w**orking **d**irectory)
* **ls** [-al] *directory* (**l**i**s**t the contents of a directory)
* **cd** *directory* (**c**hange the working **d**irectory)
* **cat** *file* (con**cat**enate files and print out the contents)
* **cp** *source destination* (**c**o**p**y files)
* **mv** *source destination* (**m**o**v**e or rename a file)
* **rm** *file* (**r**e**m**ove files)

Basic Unix/Linux Commands (cont.)

* **mkdir** *directory* (**m**a**k**e new **dir**ectories)
* **rmdir** *directory* (**r**e**m**ove **dir**ectories)
* **file** *filename* (display the type of **file**s)
* **man** [123] *command* (display the on-line **man**ual)
  * 1 > User Commands
  * 2 > System Calls
  * 3 > C Library Functions

Development Tools

- Desktop based tools
  - X Code (Needed “xcode-select --install” to install Command Line Tools)
  - Code Blocks
  - Visual Studio Code
- Web based tools
  - Support single editor and terminal (https://www.onlinegdb.com/, https://www.tutorialspoint.com/codingground.htm)
  - Support multiple editors and terminals (https://replit.com, https://sandbox.cs50.io)

Development System Map

- Programs (e.g., gcc, g++)
  - /usr/bin for general use
  - /usr/local/bin and /opt
    - added by system administrators for specific use
- Header Files (e.g., gcc –I/home/term1/include foo.c)
  - /usr/include
- Library Files (e.g., gcc –o foo -L/usr/lib foo.c libm.a)
  - /lib and /usr/lib

Editors

- **vim** [*filename*]
  - **i** (**i**nsert for entering the editing mode)
  - **esc** (**esc**ape for exiting the editing mode)
  - **w** *filename* (**w**rite for saving the contents)
  - **q** (**q**uit for exiting the vim program)

Editors (cont.)

- **nano** [*filename*]
  - **Ctrl+Shift+6** or **Ctrl+6** (on Windows) = Mark text (with right arrow to select text)
  - **Alt+Shift+6** or **Alt+6** (on Windows) or **Esc+6** (on MacOS) = Copy a block of text
  - **Ctrl+u** = Paste a block of text
  - **Ctrl+c** = Show the current line and column
  - **Ctrl+Shift+-** = Go to line and column specified
  - **Ctrl+o** = Save contents of a file
  - **Ctrl+x** = Exit the nano program

Compiler

- **gcc**
  - -**c** *source.c* (**c**ompile option for compiling source files only)
  - -**o** *target* (**o**utput option for creating the specified executable file)
  - -**g** (debu**g** option for creating the debugging information)
  - -**v** (**v**erbose option for displaying include-paths and library-paths)
  - -**Wall** (show **all** **w**arning messages)
  - -**lm** (**l**ink with the **m**ath library)
  - -**lpthread** (**l**ink with the **p**osix **thread** library)

Compiler (cont.)

- **gcc examples**
  - **gcc** *source.c* —–> *a.out*
  - **gcc** -**o** *target source.c* —–> *target*
  - **gcc** -**g** -**o** *target source.c* —–> *target* and *target.dSYM*
  - **gcc** -**Wall** -**g** -**o** *target source.c* —–> *target, target.dSYM* and *warnings*
  - **make** *target* —–> *target* and … (depends on the makefile)

Static Libraries

- Static Libraries
  - Also known as archives therefore have names ended with the **.a**
  - Can be created by using the **ar** (for archive) program
    - Example:
      - $gcc –c foo.c part1.c part2.c
      - $gcc –o foo foo.o part1.o
      - $**ar** crv **lib.a** part1.o part2.o
      - $gcc –o foo foo.c **lib.a**
      - $ranlib **lib.a** for creating the table of contents of the library

Shared Libraries

- Shared Libraries
  - Many programs use the same functions from the same static library will have **many copies of the same functions in memory and also many copies in the program files**. **The shared libraries can overcome these disadvantages**.
  - Located at the same place as static libraries; however they have a different version (e.g., /lib/ld-linux.so.N where N is a major version number of the share library.)

Shared Libraries (cont.)

- **Shared libraries consist of two basic parts**: the stub and the image. The stub library has an extension of .sa. The **stub** is the library an executable will be linked to. It **provides references to the location of the real shared functions and variables** reside in memory. The library **image contains the actual executable functions** used by binary programs and has an extension of .so, followed by a version number.

- When a program is loaded into memory to be executed, the function references are resolved. **Whenever calls are made to the shared library, the library will be loaded into the memory at run time**.

Shared Libraries (cont.)

- Another benefit of the **shared libraries** is that they **can be updated independently of the programs** that rely on it. Minor version number (revision) can be made to appear transparently to the client programs by using a symbolic link (e.g., ln –s /lib/libc.so.5.2.8 /lib/libc.so.5)
- In Linux system, the program that takes care of loading shared libraries and resolving client program function references is ld.so or ld-linux.so. The location of shared libraries are listed in the file /etc/ld.so.conf, which needs to be processed by ldconfig if changed. The utility ldd can be used to see which shared libraries are required by a program (e.g. ldd foo libc.so.5 (DLL Jump 5.2p18) => /lib/libc.so.5.2.8))

Shared Libraries (cont.)

- Example:
  - gcc –c part1.c part2.c —> (1)
  - gcc **-fpic -shared** –o **sfoo.so** part1.o part2.o —> (2)
  - gcc **-fpic -shared** –o **sfoo.so** part1.c part2.c —> (1) + (2)
  - gcc –o sfoo foo.c **sfoo.so**
  - export LD_LIBRARY_PATH=. (library path is required in some system)
  - ./sfoo

End of Module