System programming

Intro

gkagombe@jkuat.ac.ke

What is system programming?

- *System programming* is the practice of writing *system software*.
- *A program that uses system level services directly is called a system program, and the type of programming that uses these services is called system programming.*
- System programs make requests for resources and services directly from the operating system and may even access the system resources directly.

System softwares

- System software lives at a low level, interfacing directly with the kernel and core system libraries.
- Examples; shell, text editor, compiler and debugger, core utilities and system daemons. others are the network server, the web server, and the database.
- These components are entirely system software, primarily if not exclusively interfacing with the kernel and the C library.
- Other software (such as high-level GUI applications) lives at a higher level, delving into the low level only on occasion.

System vs application programming

- The system programmer must have an acute awareness of the hardware and the operating system on which they work.
- system programs interface primarily with the kernel and system libraries. application programs interface with high-level libraries. These libraries *abstract* away the details of the hardware and operating system.
  - Such abstraction has several goals: portability with different systems, compatibility with different versions of those systems, and the construction of higher-level toolkits that are easier to use, more powerful, or both.

What is UNIX?

- Originally developed at Bell Laboratories. Designed in the 1970s for Digital Equipment PDP computers.
- Become very popular multiuser, multitasking O/S for a wide variety of different hardware platforms;PC workstations, multiprocessor servers, supercomputers,etc.
- UNIX – a trademark administered by X/Open.

=>*a computer operating system that conforms to the X/Open specification XPG4.2*.

  - This specification, (aka SPEC1170), defines the names of, interfaces to and behaviors of all UNIX operating system functions.
  - It is largely a superset of an earlier series of specifications, the P1003, or POSIX specifications, actively being developed by the IEEE .

- Available UNIX−like systems :either commercially, such as Sun's Solaris for SPARC and Intel processors, or free, such as FreeBSD and Linux.
- Only few systems currently conform to the X/Open specification, which allows them to be branded UNIX98.
- Compatibility, a problem between different UNIX systems. POSIX

What is Linux?

- A freely distributed implementation of a UNIX−like kernel, the low level core of an operating system.
- Linux takes the UNIX system as its inspiration, therefore Linux and UNIX programs are very similar.
  - Almost all programs written for UNIX can be compiled and run under Linux.
  - many commercial applications sold for commercial versions of UNIX can run unchanged in binary form on Linux systems.
- Developed by Linus Torvalds at the University of Helsinki, with the help of UNIX programmers from across the Internet.
- It began as a hobby inspired by Andy Tanenbaum's Minix, a small UNIX system, but has grown to become a complete UNIX system in its own right.
- The Linux kernel doesn't use code from AT&T or any other proprietary source.

The GNU Project and the Free Software Foundation

- Commercial UNIX systems come bundled with applications programs which provide system services and tools.
- For Linux systems, these programs are written by different programmers and have been freely contributed.
- Linux community (together with others) supports free software, i.e. software that is free from restrictions, subject to the GNU General Public License. Might cost to obtain the S/W, but thereafter can be used in any way desired, usually distributed in source form.
- The Free Software Foundation - set up by Richard Stallman, the author of GNU Emacs, - one of the best known editors for UNIX and other systems.
- Stallman: a pioneer of the free software concept. started the GNU project, an attempt to create an operating system and development environment that will be compatible with UNIX. may turn out to be very different from UNIX at the lowest level, but will support UNIX applications.
- The name GNU stands for GNU's Not Unix.
- GNU software, are distributed under the terms of the GNU Public License (GPL).

- Software from the GNU Project distributed under the GPL includes:
  - GCC — A C compiler
  - G++ — A C++ compiler
  - GDB — A source code level debugger
  - GNU make — A version of UNIX make
  - Bison yacc — A parser generator compatible with UNIX
  - Bash — A command shell
  - GNU Emacs — A text editor and environment

- Other softwares are; graphical image manipulation tools, spreadsheets, source code control tools, compilers and interpreters, internet tools and a complete object−based environment: GNOME.
- http://www.gnu.org.

UNIX Programs

- Applications under UNIX are represented by two special types of file: executables and scripts.
  - Executable files: programs that can be run directly by the computer and correspond to DOS .exe files.
  - Scripts :collections of instructions for another program, an interpreter, to follow. Correspond to DOS .bat files, or interpreted BASIC programs.
- In UNIX : - executables or scripts don't require specific file name nor any particular extension. File system attributes, are used to indicate that a file is a program that may be run.
- can replace scripts with compiled programs (and vice versa) without affecting other programs or the people who call them. At the user level, no difference between the two.

- When you log in to a UNIX system, you interact with a shell program (often sh) that undertakes to run programs for you.
  - finds the programs you ask for by name: search for a file with the same name in a given set of directories.
  - Directories to search are stored in a shell variable, PATH. The search path (to which you can add) is configured by your system administrator and will usually contain some standard places where system programs are stored. These include:

/bin Binaries,	programs used in booting the system.
/usr/bin	User binaries, standard programs available to users.
/usr/local/bin	Local binaries, programs specific to an installation.

- An administrator's login, such as root, may use a PATH variable that includes directories where system administration programs are kept, such as /sbin and /usr/sbin.
- Optional operating system components and third−party applications may be installed in subdirectories of /opt,
- Installation programs might add to your PATH variable by way of user install scripts.
- It is probably a good idea not to delete directories from PATH unless you are sure that you understand what will result if you do.

Note: UNIX uses the : character to separate entries in the PATH variable.

Example

PATH variable:

/usr/local/bin:/bin:/usr/bin:.:/home/neil/bin:/usr/X11R6/bin

- Here the PATH variable contains entries for the standard program locations, the current directory (**.**), a user's home directory and the X Window System.

The C Compiler

Our First UNIX C Program

- Here's the source code for the file hello.c:

```
#include <stdio.h>
int main()
{
printf("Hello World\n");
exit(0);
}
```

1. To enter this program, you'll need to use an editor. E.g. the vi editor, emacs, etc..
   - learn emacs, after starting it - press *Ctrl−H*, followed by *t* for the tutorial. has its entire manual available on−line. Try *Ctrl−H* and then *i* for information.
   - Some versions may have menus, used to access the manual and tutorial.
2. On POSIX−compliant systems, the C compiler is called C89. Historically, the C compiler was simply called cc. Over the years, different vendors have sold UNIX−like systems with C compilers with different facilities and options, but often still called cc.

- the POSIX committee decided to create a new standard command for the C compiler, c89. When this command is present, it will always take the same options, independent of the machine.
- On Linux systems, you might find that any or all of the commands c89, cc and gcc refer to the system C compiler, usually the GNU C compiler. On UNIX systems, the C compiler is almost always called cc.
- We will be using GNU C, because it's provided with Linux distributions and because it supports the ANSI standard syntax for C.
- If you're using a UNIX system without GNU C, you can obtain and install it starting at http://www.gnu.org.
- To compile, link and run our program.

```
$ cc –o hello hello.c
$ ./hello
Hello World
$
```

How It Works

- We invoked the system C compiler which translated our C source code into an executable file called hello.
- We ran the program and it printed a greeting.

Note:

- The hello program will probably be in your home directory. If PATH doesn't include a reference to your home directory, the shell won't be able to find hello.
- another program called hello will be executed instead:- if it is contained in one of the directories in PATH(and home dir not included in PATH). – or if such a directory is mentioned in PATH before your home directory.
- To get around this , prefix program names with ./ (e.g. ./hello). specifically instructs the shell to execute the program in the current directory with the given name.
- If you forget the −o name option (tells the compiler where to place the executable) the compiler will place the program in a file called a.out (meaning assembler output).
- *http://www.tutorialspoint.com/execute_bash_online.php*

Getting Help

- All UNIX systems are reasonably well−documented with respect to the system programming interfaces and standard utilities.
- UNIX system programmers supply a manual page with their programs. These manual pages, which are sometimes provided in a printed form, are invariably available online.
- The man command provides access to the online manual pages.
- The GNU software suite and some other free software uses an online documentation system called info.
- You can browse full documentation online using a special program, info, or via the info command of the emacs editor.
- The benefit of the info system; navigate the documentation using links and cross−references to jump directly to relevant sections. For the documentation author, the info system has the benefit that its files can be automatically generated from the same source as the printed, typeset documentation.

Try It Out – Manual Pages and info

- Let's look for documentation of the GNU C compiler. First, the manual page.

  $ **man gcc**

- When reading manual pages you can use the spacebar to read the next page, *Return* to read the next line and *q* to quit altogether.
- To get more information on GNU C, we can try info.

  $ **info gcc**

- We're presented with a long menu of options that we can select to move around a complete text version of the documentation. Menu items and a hierarchy of pages allow us to navigate a very large document.
- On paper, the GNU C documentation runs to many hundreds of pages.
- The info system also contains its own help page in info form pages. If you type *Ctrl−H,* you'll be presented with some help which includes a tutorial on using info.
- The info program is available with many Linux distributions and can be installed on other UNIX systems.

Standards

- To help bring order to chaos, standards groups codify system interfaces into official standards.
- Numerous such standards exist but, technically Linux does not officially comply with any of them.
- Linux *aims* toward compliance with two of the most important and prevalent standards: POSIX and the Single UNIX Specification (SUS).
- POSIX and SUS document, among other things, the C API for a Unix-like operating system interface. Effectively, they define system programming, or at least a common subset thereof, for compliant Unix systems.

POSIX and SUS History

- In the mid-1980s, the Institute of Electrical and Electronics Engineers (IEEE) spearheaded an effort to standardize system-level interfaces on Unix systems.
- Richard Stallman, suggested the standard be named *POSIX* which now stands for *Portable Operating System Interface*.
- The first result of this effort, issued in 1988, was IEEE Std 1003.1-1988 (POSIX 1988, for short).
- In 1990, the IEEE revised the POSIX standard with IEEE Std 1003.1-1990 (POSIX 1990).
- Optional real-time and threading support were documented in, respectively, IEEE Std 1003.1b-1993 (POSIX 1993 or POSIX.1b), and IEEE Std 1003.1c-1995 (POSIX 1995 or POSIX.1c).
- In 2001, the optional standards were rolled together with the base POSIX 1990, creating a single standard: IEEE Std 1003.1-2001 (POSIX 2001).
- The latest revision, released in December 2008, is IEEE Std 1003.1-2008 (POSIX 2008).
- All of the core POSIX standards are abbreviated POSIX.1, with the 2008 revision being the latest.

Cont…

- In the late 1980s and early 1990s, Unix system vendors were engaged in the “Unix Wars,” with each struggling to define its Unix variant as *the* Unix operating system.
- Several major Unix vendors rallied around The Open Group, an industry consortium formed from the merging of the Open Software Foundation (OSF) and X/Open.
- The Open Group provides certification, white papers, and compliance testing.
- In the early 1990s, with the Unix Wars raging, The Open Group released the Single UNIX Specification (SUS).
- SUS rapidly grew in popularity, in large part due to its cost (free) versus the high cost of the POSIX standard.
- Today, SUS incorporates the latest POSIX standard.
- The first SUS was published in 1994. revisions in 1997 (SUSv2) and 2002 (SUSv3). The latest SUS, SUSv4, was published in 2008. SUSv4 revises and combines IEEE Std 1003.1-2008 and several other standards.

C Language Standards

- Dennis Ritchie and Brian Kernighan's book, *The C Programming Language* (Prentice Hall, 1978), acted as the informal C specification for many years (*K&R C).*
- C was rapidly replacing BASIC and other languages in popularity, therefore, the need to officially standardize.
- 1983 ANSI formed a committee to develop an official version of C. *ANSI C* was completed in 1989.
- In 1990, *ISO C90*, based on ANSI C, with a handful of modifications.
- In 1995, *ISO C95 was* released- an update, rarely implemented
- In 1999, *ISO C99* - large update to the language, introduced many new features, including inline functions, new data types, variable-length arrays, C++-style comments, and new library functions.
- The latest version is *ISO C11*-most significant feature; a formalized memory model, enables the portable use of threads across platforms.

On the C++ front, ….

- ISO standardization was slow in arriving.
- the first C standard, ISO C98, was ratified in 1998. While it greatly improved compatibility across compilers, several aspects of the standard limited consistency and portability.
- *ISO C++03* arrived in 2003.- offered bug fixes to aid compiler developers but no user-visible changes.
- The next and most recent ISO standard, *C++11* (formerly *C++0x* in suggestion of a more optimistic release date), heralded numerous language and standard library additions and improvements—so many, in fact, that many commentators suggest C++11 is a distinct language from previous C++ revisions.

Linux and the Standards

- Linux aims toward POSIX and SUS compliance. It provides the interfaces documented in SUSv4 and POSIX 2008, including real-time (POSIX.1b) and threading (POSIX.1c) support.
- Linux strives to behave in accordancewith POSIX and SUS requirements.

  => *Failing to agree with the standards is considered a bug*.

Note: Linux is believed to comply with POSIX.1 and SUSv3, but no official POSIX or SUS certification has been performed (particularly on each and every revision of Linux), we cannot say that Linux is officially POSIX- or SUS-compliant.

- With respect to language standards, Linux fares well:
  - The *gcc* C compiler is ISO C99-compliant; support for C11 is ongoing.
  - The *g++* C++ compiler is ISO C++03-compliant with support for C++11 in development.
  - *Also gcc* and *g++_* implement extensions to the C and C++ languages. These extensions are collectively called *GNU C*.

Concepts of Linux Programming

- All Unix systems, Linux included, provide a mutual set of abstractions and interfaces. These commonalities *define* Unix.
- Abstractions such as the file and the process, interfaces to manage pipes and sockets, and so on, are at the core of a Unix system.
- This is an overview of the foundation of Linux system programming.

Files and the Filesystem

- File: most basic and fundamental abstraction in Linux.
- Linux follows the *everything-is-a-file* philosophy .
- Consequently, much interaction occurs via reading of and writing to files, even when object in question is not a normal file.
- Must first be opened, in order to be accessed. Can be opened for reading, writing, or both.
- An open file is referenced via a unique descriptor, a mapping from the metadata associated with the open file back to the specific file itself.
- Inside the Linux kernel, this descriptor is handled by an integer (of the C type int) called the *file descriptor*(*fd)*.
- Fd's are shared with user space, and are used directly by user programs to access files.

Regular files

- contains bytes of data, organized into a linear array -a byte stream.
- In Linux, no further organization or formatting is specified for a file. The bytes may have any values, and they may be organized within the file in any way.
- At the system level, Linux does not enforce a structure upon files beyond the byte stream. Some operating systems, such as VMS, provide highly structured files, supporting concepts such as *records*. Linux does not.
- Any of the bytes within a file may be read from or written to.
- These operations start at a specific byte, which is one's conceptual "location" within the file. This location is called the *file position* or *file offset*.
  - The file position ; an essential piece of the metadata that the kernel associates with each open file.
  - When a file is first opened, the file position is zero. as bytes in the file are read from or written to, byte-by-byte, the file position increases in kind.

  - may also be set manually to a given value, even a value beyond the end of the file.
  - starts at zero; it cannot be negative.
- Writing a byte to the middle of a file overwrites the byte previously located at that offset.=> not possible to expand a file by writing into the middle of it. Most file writing occurs at the end of the file.
  - maximum value is bounded only by the size of the C type used to store it.
- The size of a file is measured in bytes and is called its *length*.

=>Length- the number of bytes in the linear array that make up the file.

- The maximum file length, is bounded only by limits on the sizes of the C types that the Linux kernel uses to manage files.

- A single file can be opened more than once, by a different or even the same process.
- Each open instance of a file is given a unique file descriptor.
- processes can share their file descriptors.
  - The kernel does not impose any restrictions on concurrent file access.
  - Multiple processes are free to read from and write to the same file at the same time.
- files are usually accessed via *filenames*, but they actually are not directly associated with such names.
- Instead, a file is referenced by an *inode* (originally short for *information node*),
  - assigned an integer value unique to the filesystem -*inode number* (*i-number* or *ino)* (not necessarily unique across the whole system).
  - stores metadata associated with a file, such as its modification timestamp, owner, type, length, and the location of the file's data—but no filename!
- The inode is both a physical object, located on disk in Unix-style filesystems, and a conceptual entity, represented by a data structure in the Linux kernel.

Directories and links

- Accessing a file via its inode number is cumbersome (and also a potential security hole), so files are always opened from user space by a name, not an inode number.
- *Directories* are used to provide the names with which to access files.
  - acts as a mapping of human-readable names to inode numbers.
  - A name and inode pair is called a *link*.

  The physical on-disk form of this mapping—for example, a simple table or a hash—is implemented and managed by the kernel code that supports a given filesystem.
- Conceptually, a directory is a normal file, that contains only a mapping of names to inodes. The kernel directly uses this mapping to perform name-to-inode resolutions.
- When a user-space application requests that a given filename be opened;
  - kernel opens the directory containing the filename and searches for the given name.
  - From the filename, the kernel obtains the inode number.
  - From the inode number, the inode is found. The inode contains metadata associated with the file, including the on-disk location of the file's data.

- Initially, there is only one directory on the disk, the *root directory*. denoted by the path */*.

Note: the links inside of directories can point to the inodes of other directories. i.e. directories can nest inside of other directories, forming a hierarchy of directories. allowing for the use of the *pathnames*—for example, */home/blackbeard/concorde.png*.

- When kernel is asked to open a pathname like this, it walks each *directory entry* (called a *dentry* inside of the kernel) in the pathname to find the inode of the next entry.
- In the example, the kernel starts at */*, gets the inode for *home*, goes there, gets the inode for *blackbeard*, runs there, and finally gets the inode for *concorde.png*.
- This operation is called *directory* or *pathname resolution*.
- Linux kernel also employs a cache, called the *dentry cache*, -store the results of directory resolutions, providing for speedier future lookups.
- *absolute pathname: is a fully qualified* i.e starts at the root directory.
- *relative pathnames:* are provided relative to some other directory (for example, *todo/plunder*).
  - the kernel begins the pathname resolution in the *current working directory*.

- Although directories are treated like normal files, the kernel does not allow them to be opened and manipulated like regular files.
- they must be manipulated using a special set of system calls that only allow for the adding and removing of links(only sensible operations).
- If user space were allowed to manipulate directories without the kernel's mediation, it would be too easy for a single simple error to corrupt the filesystem.

Hard links

- When multiple links map different names to the same inode, we call them *hard links*.
- Hard links allow for complex filesystem structures with multiple pathnames pointing to the same data.
- The hard links can be in the same directory, or in two or more different directories.
- For example, a specific inode that points to a specific chunk of data can be hard linked from */home/bluebeard/treasure.txt* and */home/blackbeard/to_steal.txt*.
- Deleting a file involves *unlinking* it from the directory structure, which is done simply by removing its name and inode pair from a directory.
- To ensure that a file is not destroyed until *all* links to it are removed, each inode contains a *link count (*keeps track of the number of links within the filesystem that point to it). When a pathname is unlinked, the link count is decremented by one;
- only when it reaches zero are the inode and its associated data actually removed from the filesystem.

Symbolic links

- Hard links cannot span filesystems because an inode number is meaningless outside of the inode's own filesystem.
- *symbolic links* (*symlinks*)- allow links that can span filesystems, and that are a bit simpler and less transparent,
- look like regular files. has its own inode and data chunk, which contains the complete pathname of the linked-to file.
  - meaning symbolic links can point anywhere, including to files and directories that reside on different filesystems,
  - even to files and directories that do not exist. A symbolic link that points to a nonexistent file is called a *broken link*.

Comparison…

- Symbolic links incur more overhead than hard links - resolving a symbolic link effectively involves resolving two files: the symbolic link and then the linked-to file.

Hard links do not incur this additional overhead—no difference between accessing a file linked into the filesystem more than once and one linked only once.

  - The overhead of symbolic links is minimal, but it is still considered a negative.

- Symbolic links are also more opaque than hard links. Using hard links is entirely transparent; in fact, it takes effort to find out that a file is linked more than once! Manipulating symbolic links, requires special system calls.
  - This lack of transparency is often considered a positive, as the link structure is explicitly made plain, with symbolic links acting more as *shortcuts* than as filesystem-internal links.

Special files

- kernel objects that are represented as files. A way to let certain abstractions fit into the filesystem, continuing the everything-is-a-file paradigm.
- Linux supports four:
  - *block device files, character device files, named pipes, and Unix domain sockets.*
- Device files: act and look like normal files residing on the filesystem.
- may be opened, read from, and written to, allowing user space to access and manipulate devices (both physical and virtual) on the system.
- generally broken into two groups: *character devices* and *block devices*.
- Each type of device has its own special device file.
- A character device:
  - accessed as a linear queue of bytes. The device driver places bytes onto the queue, one by one, and user space reads the bytes in the order that they were placed on the queue.
  - E.g. A keyboard.
  - When there are no more characters left to read, the device returns end-of-file (EOF).
- Character devices are accessed via *character device files*.
- A block device,:
  - accessed as an array of bytes.
  - The device driver maps the bytes over a seekable device, and user space is free to access any valid bytes in the array, in any order—it might read byte 12, then byte 7, and then byte 12 again.
  - They are generally storage devices. E.g. Hard disks, floppy drives, CD-ROM drives, and flash memory
- They are accessed via *block device files*.

- *Named pipes* (*FIFOs*, short for "first in, first out") are an *interprocess communication* (*IPC*) mechanism that provides a communication channel over a file descriptor, accessed via a special file.
- Regular pipes are the method used to "pipe" the output of one program into the input of another; they are created in memory via a system call and do not exist on any filesystem.
- Named pipes act like regular pipes but are accessed via a file, called a *FIFO special file*. Unrelated processes can access this file and communicate.

Sockets :

- an advanced form of IPC that allow for communication between two different processes, not only on the same machine, but even on two different machines.
- form the basis of network and Internet programming.
- come in multiple varieties.
- the Unix domain socket, which is the form of socket used for communication within the local machine. use a special file residing on a filesystem, often simply called a socket file.
- sockets communicating over the Internet might use a hostname and port pair for identifying the target of communication.