Introduction to Unix

Columbia College

Roman Bodrykh

February 27, 2018

Lecture Outline

- Advantages of using Linux over Windows
- Different computer system
- Structure of the Unix System
- Secure Shell (SSH)
- The UNIX filesystem and directory structure
- Users and Permissions
- Commands. File and directory handling commands.
- Install Linux Mint in a virtual machine
  - try superuser powers
  - Installing applications

Advantages of using Linux over Windows

- Linux is Free and Faster in Performance than Windows
- Fast Booting
    - You can see Linux Mint, Ubuntu, and Linux distros booting faster than any Windows
- Lightweight
    - Windows need 15 to 20 Gb space for installation requirement while Linux need only 2 to 3 Gb of space for total installation
- Linux OS will be Perfect for low Hardware PC's and Laptops
- User File Security
    - All files are encrypted and system files are password protected
    - Every modifications and updates need user confirmation
- One Click Update
    - All system files and applications can be updated with single click update
- Linux Suits Best for Programmers
- Linux can be easily Customisable for the User

Different computer system - Server and Workstation

A server manages all network resources. Servers are often dedicated (meaning it performs no other task besides servertasks)

A desktop computer system typically runs a user-friendly operating system and desktop applications to facilitate desktop-oriented tasks

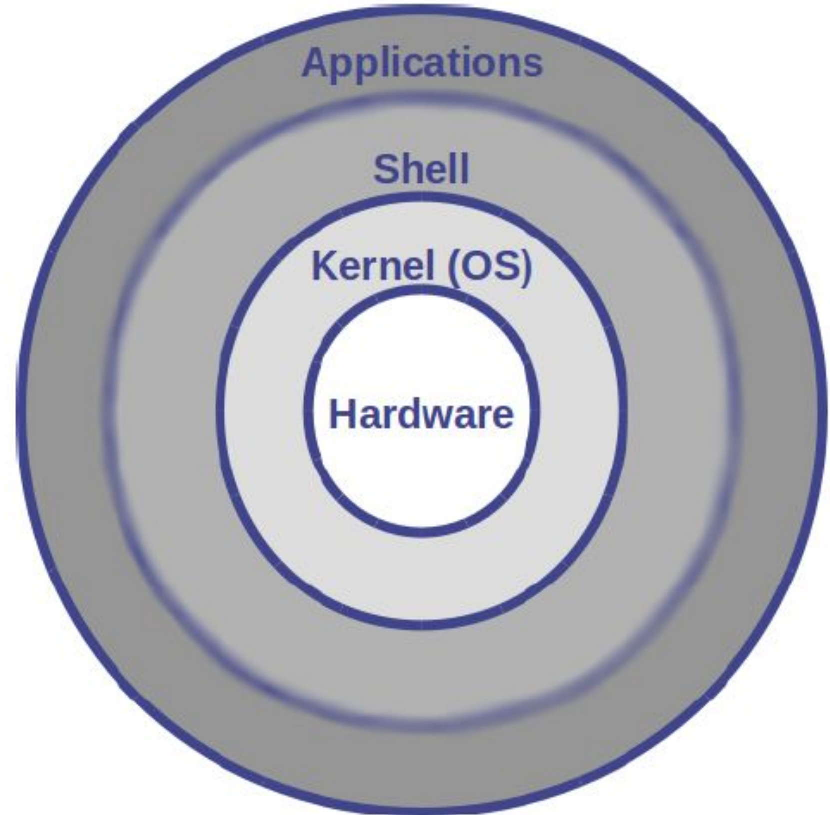
Different computer system - Server vs Workstation (PC)

Server	PC
Uses Xeon or Itanium processor in multiple numbers	Uses single desktop processor
Multiple Hard disks are used	Single hard disk is used
Higher memory capacity and in multiple numbers e.g. 2 or 4 or 8	Lower memory capacity and less in number e.g. 1 or 2.
Network capability is better	Normal network capability
Serves more number of users e.g. 100 or 500 or 1000	Are built for individual user
Structure is more advanced and complex	Simple structure

Structure of the Unix System

There are two important divisions in UNIX operating system architecture:

- Kernel – interacts with the machine's hardware
- Shell – interacts with the user



Structure of the Unix System - Kernel

- Interacts directly with the hardware through device drivers
- Provides sets of services to programs, insulating these programs from the underlying hardware
- Manages memory, controls access, maintains filesystem, handles interrupts, allocates resources of the computer

SOME OTHER FUNCTIONS PERFORMED BY THE KERNEL IN UNIX SYSTEM ARE:

- Scheduling the work done by the CPU so that the work of each user is carried out as efficiently as is possible
- Organizing the transfer of data from one part of the machine to another
- Accepting instructions from the unix shell and carrying them out
- Enforcing the access permissions that are in force on the file system

Structure of the Unix System - Shells and GUIs

- Helps manage user applications
- An interactive shell is the user interface (responds to user commands)
- A desktop is a GUI shell
- A shell is just another program
- UNIX Shell acts as a medium between the user and the kernel in unix system. When a user logs in, the login program checks the username and password and then starts another program called the shell.
- The commands are themselves programs: when they terminate, the shell gives the user another prompt (% on our systems).
- Even though there is only one kernel running on the unix system, there could be several shells in action – for each user who is logged in.
- The shell keeps a list of the commands you have typed in. If you need to repeat a command, use the cursor keys to scroll up and down the list or type history for a list of previous commands.

Structure of the Unix System - System Utilities

- Includes commands such as **ls**, **cp**, **mv**, **rm**, **grep**, **awk**, **sed**, **bc**, **wc**, **more**, and so on

- These system utilities are designed to be powerful tools that do a single task extremely well (e.g. **grep** finds text inside files while wc counts the number of words, lines and bytes inside a file)

- Users can often solve problems by interconnecting these tools instead of writing a large monolithic application program

Structure of the Unix System - System Utilities

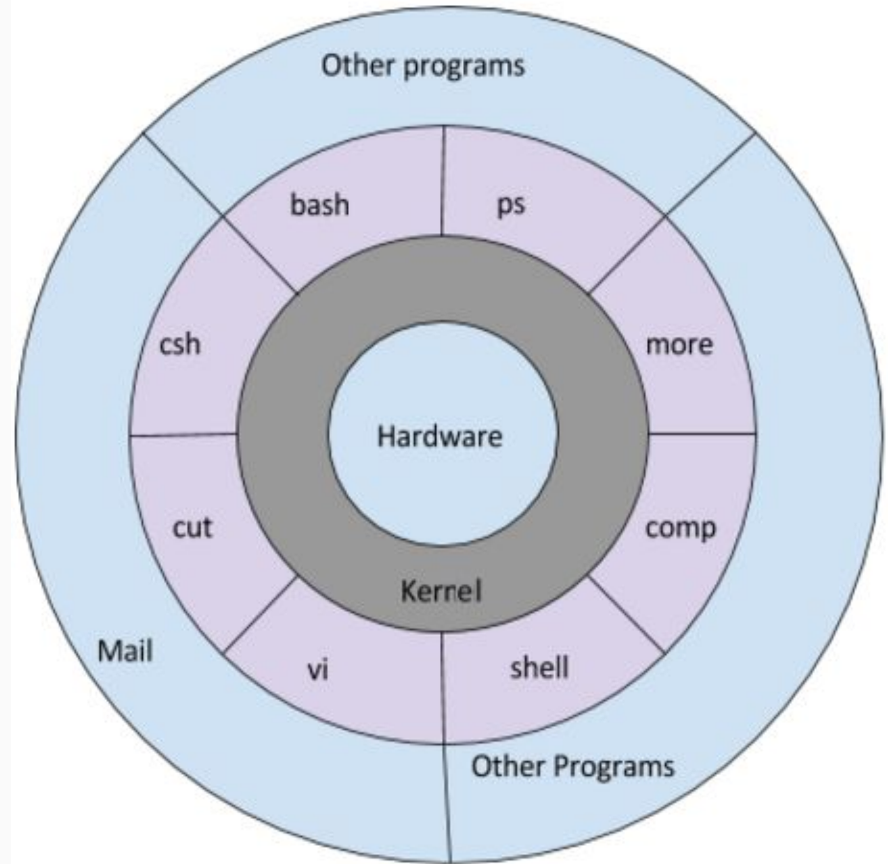
Like other UNIX flavours, Linux's system utilities also include server programs called **daemons** which provide remote network and administration services

- **telnetd** and **sshd** provide remote login facilities
- **lpd** provides printing services
- **Httpd** serves web pages
- **crond** runs regular system administration tasks automatically

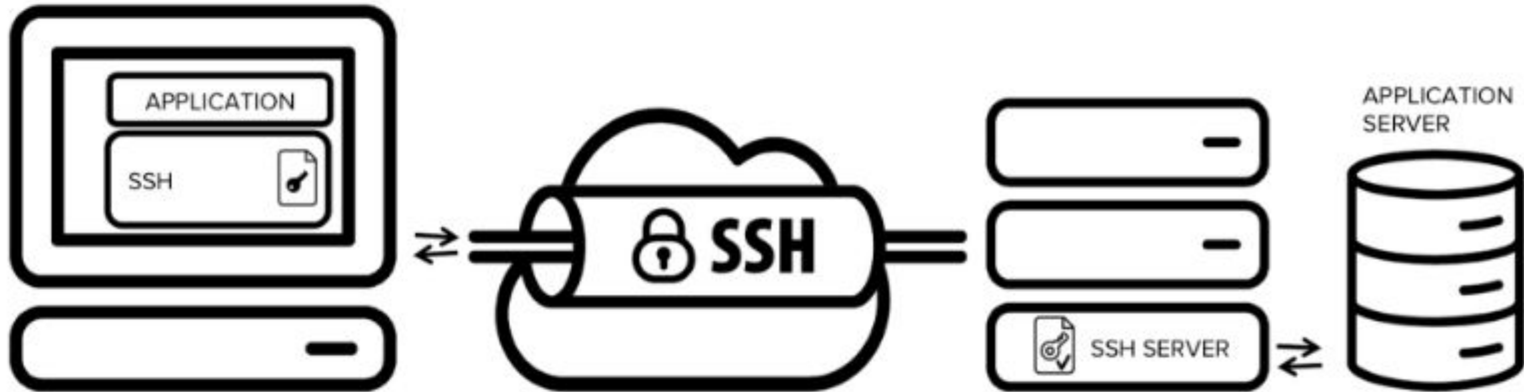
Structure of the Unix System - Applications

There are many standard applications:

- Filesystem commands
- Text editors
- Compilers
- Text processing



Secure Shell (SSH) - Secure Remote Access - What is SSH?



- SSH – Secure Shell
- SSH is a protocol for secure remote login and other secure network services over an insecure network
- developed by SSH Communications Security Corp., Finland
- two distributions are available:
  - commercial version
  - freeware (www.openssh.com)

Secure Shell (SSH) About SSH

- SSH is both a program and a protocol
    - Allows users to securely log into another computer over an insecure network, executes commands and transfers files
    - Created as a replacement for TELNET, ftp, and rlogin, rsh, and rcp
    - Uses TCP and provides authentication, confidentiality (both data and command), integrity, authorization, data compression, and with SSH-2, multiplexing
    - Has transparent client/server communication over encrypted network connections
    - Can be implemented on most Operating Systems (Win, Mac, Unix/Linux)

Secure Shell (SSH) - Features

- **Authentication**
  - Proof of identity of users and servers, typically password and public-key signature, but other methods are available
- **Privacy**
  - Via strong standard encryption algorithms
- **Integrity**
  - Cryptographic integrity checking via MD5 and SHA-1 keyed hash algorithms
- **Authorization / Access**
  - Server configurable access
- **Forwarding or Tunnelling**
  - Encrypt other TCP/IP-based sessions
- **Data Compression**

Secure Shell (SSH) - Connection setup

- TCP connection setup
  - the server listens on default port 22 used over TCP/IP
  - the client initiates the connection

Secure Shell (SSH) - Advantages

- SSH is available on most platform
  - Clients are available for many platforms (besides major Operating System – OS/2, BeOS, Java, etc.)

- Free for noncommercial use
  - The open source version has gone through many improvements with patches, bug fixes, and addition of functionalities.
  - lsh is the General Public License (GPL) version of SSH-2 – currently being standardized by the IETF SECSH working group.

- SSH can multiplex services over the same connection
  - One of the most powerful function of multiplexing is port forwarding or tunneling
  - SSH can securely tunnel insecure applications like POP3, SMTP, IMAP, and CVS.

Secure Shell (SSH) - Disadvantages

- Only support known port number
  - Dynamic port not supported
  - Port Number can be exploited.
- SSH cannot fix all TCP's problems since TCP run below SSH
  - Can minimize attack types with authentication and security
  - Network hijacking – SSH is vulnerable to DoS
- SSH cannot protect users from attack made through other protocols.
  - E.g. NFS mounting can allow malicious access to root on UNIX/LINUX systems
- SSH provides no protection against Trojan horses or viruses

Secure Shell (SSH) - Keys and Agents

- Need for Public-Key Authentication:
  - Passwords have several drawbacks
    - Good passwords must be random/long – hard to memorize !
    - Passwords sent on network may be intercepted
    - Password changes must be communicated
  - Keys are more secure than passwords !

- What is a Key ?
  - Digital Identity (sequence of bits)
  - SSH uses a private and public key
  - Private key (client) vs Public key (server) = key pair
  - Challenge and Authenticator

Secure Shell (SSH) - Keys and Agents

- Generating Key pairs:
  - ssh-keygen creates a public and private key
  - A pass-phrase is supplied to protect the private key
  - OpenSSH can use either the RSA or DSA algorithm
  - Public key and private key are stored on the local machine after they are mathematically generated
    - ~/.ssh (SSH1/OpenSSH) or ~/.ssh2 (SSH2)
    - Private key SSH1 identity
    - Public key SSH1 identity.pub
    - Private key SSH2 id_dsa_1024_a
    - Public key SSH2 id_dsa_1024_a.pub
  - Private key is encrypted by pass-phrase and is only viewable by the person that generated it.
  - SSH2 allows a collection of private keys

Secure Shell (SSH) - Functionality

- Copy files (scp & sftp)
- **Remote terminal (ssh, slogin)**
- Remote Commands (ssh)
- Keys and agents
- Port Forwarding and Xforwarding
- SOCKS - Proxies

Secure Shell (SSH) - Remote Terminal

- Secure channel between client and server is established
  - Password supplied by client is encrypted
  - Password is sent over the network to the server
  - Server then checks the password and allows login
  - Data exchange between the two parties is secure

Secure Shell (SSH) - Remote Terminal Example

- To log into an account with the <username> on the remote computer bodrykh.net, use this command:

  $ ssh -p 59481 <username>@bodrykh.net

- The command invokes the ssh client on the local computer which contacts the ssh server running on **bodrykh.net** and asks to be logged in as **<username>**
- The following message may be seen if the SSH client encounters a new remote machine.

  Host key not found from the list of known hosts.

  Are you sure you want to continue connecting (yes/no)?

- If the user responds with a yes, the client continues:

  Host 'bodrykh.net' added to the list of known hosts.

- The known hosts database can be found at $HOME/.ssh/known_hosts

The UNIX Filesystem

The UNIX operating system is built around the concept of a filesystem which is used to store all of the information that constitutes the long-term state of the system. This state includes

- the operating system kernel itself
- the executable files for the commands supported by the operating system
- configuration information, temporary workfiles
- user data
- various special files that are used to give controlled access to system hardware and operating system functions

The UNIX Filesystem - Four types of items

Every item stored in a UNIX filesystem belongs to one of four types:

- **1. Ordinary files**
  Ordinary files can contain text, data, or program information. Files cannot contain other files or directories. Unlike other operating systems, UNIX filenames are not broken into a name part and an extension part
  Instead they can contain any keyboard character except for '/' and be up to 256 characters long (note however that characters such as *,?,# and & have special meaning in most shells and should not therefore be used in filenames). Putting spaces in filenames also makes them difficult to manipulate - rather use the underscore '_'.
- **2. Directories**
  Directories are containers or folders that hold files, and other directories.
- **3. Devices**
  To provide applications with easy access to hardware devices, UNIX allows them to be used in much the same way as ordinary files. There are two types of devices in UNIX - block-oriented devices which transfer data in blocks (e.g. hard disks) and character-oriented devices that transfer data on a byte-by-byte basis (e.g. modems and dumb terminals).
- **4. Links**
  A link is a pointer to another file. There are two types of links - a hard link to a file is indistinguishable from the file itself. A soft link (or symbolic link) provides an indirect pointer or shortcut to a file. A soft link is implemented as a directory file entry containing a pathname.

The UNIX Filesystem - Files and File Names

- A file is a basic unit of storage
- Every file has name
- Filenames are case sensitive
- Unix filenames can contain any character except the slash (/) and the null character (^@)

The UNIX Filesystem - File Names

- Every file has at least one name
- Each file in the same directory must have a unique name
- Files in different directories can have identical names
- Files that start with a (.) are by default hidden by ls, and other utilities

The UNIX Filesystem - Directories

- Sometimes called a folder
- A directory is a special sort of file which holds information about other files
- Other file types include symbolic links (just like shortcuts), named pipes, block special files (disks, USB drives)

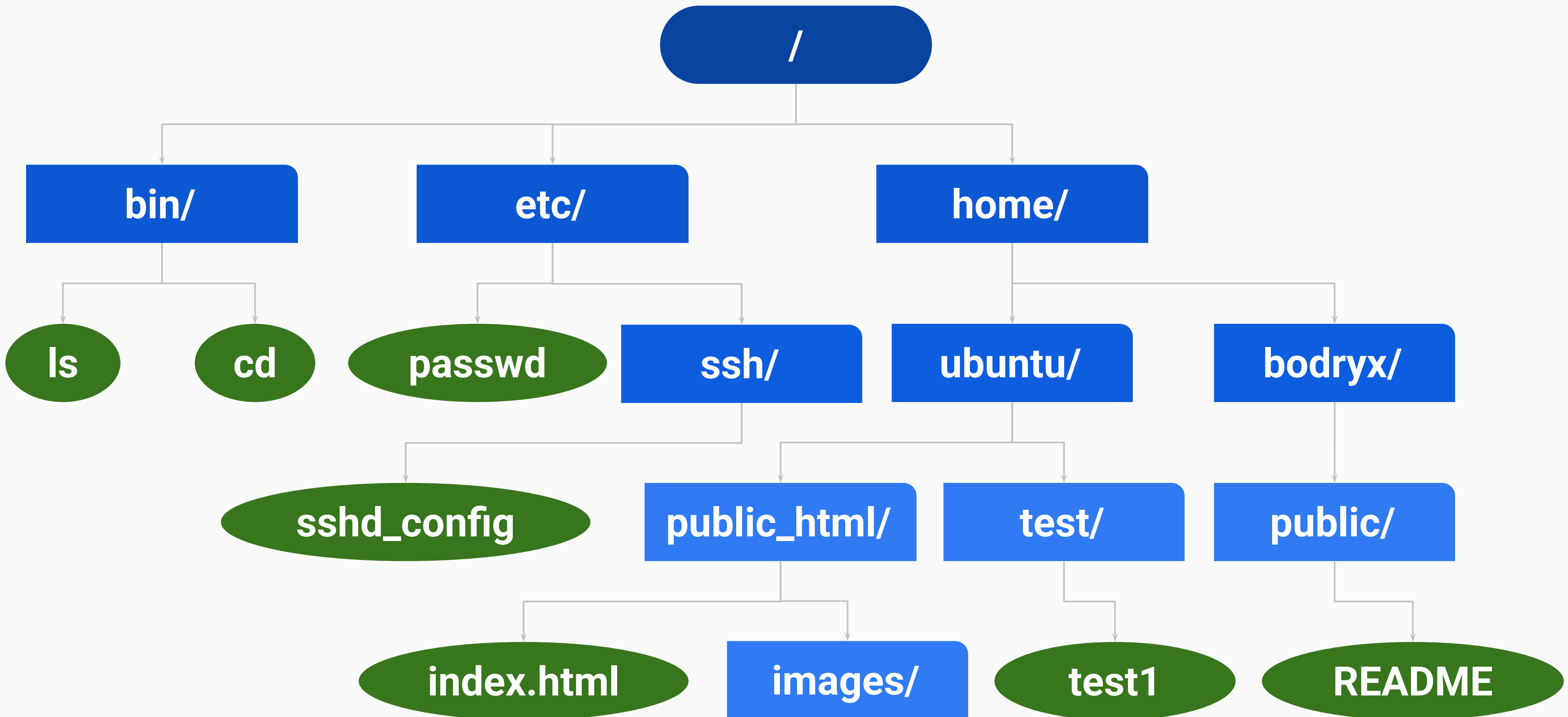
The UNIX Filesystem - Hierarchy

- A hierarchical system of organising files and directories

- The top level in the hierarchy is called the root, holds all files and directories in the filesystem its name is /

- Filesystem may span many disks, even across a network

The UNIX Filesystem - FIlesystem example



The UNIX Filesystem - Main directories - Typical Content

- **/bin** is a place for most commonly used terminal commands, like ls, mount, rm, etc.(Essential low-level system utilities)
- **/boot** contains files needed to start up the system, including the Linux kernel, a RAM disk image and bootloader configuration files.
- **/dev** contains all device files, which are not regular files but instead refer to various hardware devices on the system, including hard drives.
- **/etc** contains system-global configuration files, which affect the system's behavior for all users.
- **/home** home sweet home, this is the place for users' home directories.
- **/lib** contains very important dynamic libraries and kernel modules
- **/media** is intended as a mount point for external devices, such as hard drives or removable media (floppies, CDs, DVDs).
- **/mnt** is also a place for mount points, but dedicated specifically to "temporarily mounted" devices, such as network filesystems.
- **/opt** can be used to store additional software for your system, which is not handled by the package manager.
- **/proc** is a virtual filesystem that provides a mechanism for kernel to send information to processes.
- **/root** is the superuser's home directory, not in /home/ to allow for booting the system even if /home/ is not available.
- **/sbin** contains important administrative commands that should generally only be employed by the superuser.
- **/srv** can contain data directories of services such as HTTP (/srv/www/) or FTP.
- **/sys** is a virtual filesystem that can be accessed to set or obtain information about the kernel's view of the system.
- **/tmp** is a place for temporary files used by applications.
- **/usr** contains the majority of user utilities and applications, and partly replicates the root directory structure, containing for instance, among others, /usr/bin/ and /usr/lib.(Higher-level system utilities and application programs)
- **/var** is dedicated to variable data, such as logs, databases, websites, and temporary spool (e-mail etc.) files that persist from one boot to the next. A notable directory it contains is /var/log where system log files are kept.

The UNIX Filesystem - Home Directory

- The user's personal directory

- All home (users') directories in Unix are in **/home** (**/home/username**)

- Your current working directory (CWD) when you log in

- **cd ~** (tilde) takes you home

(Location of many startup and customisation files: **.bashrc .vimrc .forward .plan**)

The UNIX Filesystem - Pathnames

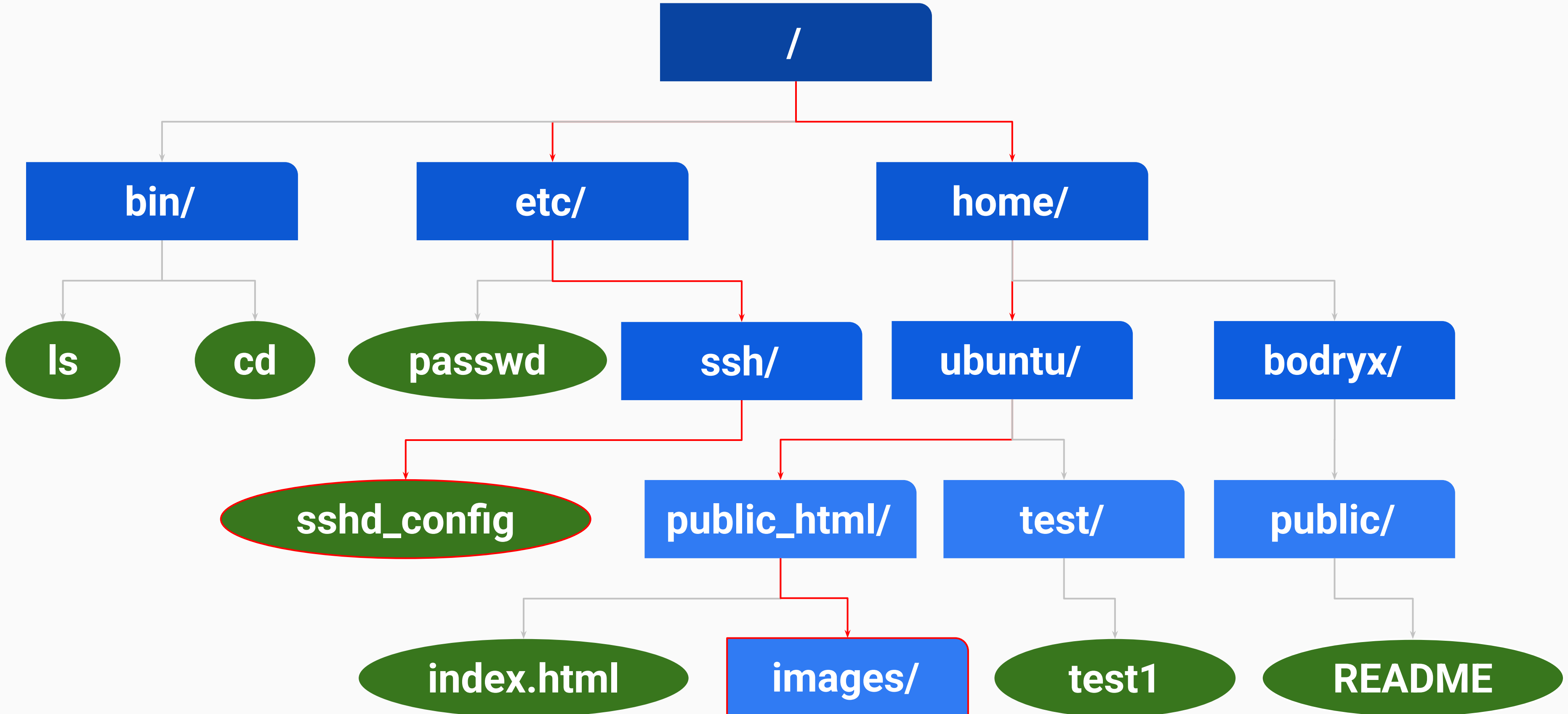
- The pathname of a file includes the name of the file, the directory that holds the file, the directory that holds that directory... up to the root

- The pathname of every file in a given filesystem is unique

- Absolute pathnames start at the root, drill down through successive subdirectories

- The forward slash, /, separates path components

The UNIX Filesystem - Pathname examples



The UNIX Filesystem - Absolute Pathnames

/home/ubuntu/public_html/images/

/etc/ssh/sshd_config

- The pathnames, above, are absolute pathnames

- An absolute path gives the location to a file or folder starting at / (the root directory)

- Uniquely identify files

The UNIX Filesystem - Relative Pathnames

- A relative path gives the location to a file or folder beginning at the current directory
- Prefixed w/the current directory, **$pwd**
- So, relative to the current directory
- Typing **cd lib** from the / directory sends you to **/lib**. From the **/usr** directory, typing **cd lib** sends you to **/usr/lib**

Example:

```
$ cd /home/bodryx
$ ls public/
index.html images
$ ls var
ls: cannot access 'var': No such file or directory
$ cd /
$ ls var
backups  cache  crash  lib  local  lock  log  mail  opt  run  snap  spool  tmp  www
```

The UNIX Filesystem - Special Relative Paths Shortcuts

- . - the current directory
- .. - the parent directory of the current directory
- ~ - Current user's (your) home directory, or /home/<username>

If we start in **/usr/local/src...**

- **~ => /home/<username>**
- **. => /usr/local/src**
- **.. => /usr/local**

$ cd ~bodryx
$ pwd
/home/bodryx

Users and Permissions - Users

Unix was designed to allow multiple people to use the same machine at once. This raises some security issues though - how do we keep our coworkers from reading our email, browsing our photo albums, etc?

- Rather than allowing everyone full access to the same files, access can be restricted to certain users' accounts.
- All accounts are presided over by the **Superuser**, or "**root**", account
- Each user has absolute control over any files he/she owns, which can only be superseded by root

Users and Permissions - Groups

Files are also assigned to groups of users, allowing certain modifications to be performed only by members of that group.

For Example:

If each member of this class had an account on the same server, it would be wise to keep your assignments private - that is a user-based restriction. However, if there were a class wiki hosted on the server, we would want everyone in this class to be able to edit it, but nobody outside this class. That situation would require all of our user accounts to belong to the same group.

Users and Permissions - File Ownership

- Each file is assigned to a single user and a single group. Ownership is usually written **user:group**
- **For example**, your files will typically belong to **yourname:users.** Root's files belong to **root:root**
- Generally it is up to **root** to change file ownership, as a regular user can't take ownership of someone else's files, and they can't pass ownership of their files to another user (or to a group they don't belong to)

Users and Permissions - Discovering Permissions

A familiar command can tell us about the ownership and permissions of files.

$ ls -l [file/dir]

- -l - lists file/directory info in a long format
- Can pass **ls** a different directory, or it defaults to

File permissions usually look something like this:

-rwxrwxrwx 1 bodryx bodryx 866 Feb 25 06:47 .bash_history

Users and Permissions - Format Example

User Group Other	number of links or directories inside	The owner that file belongs to	The group that file belongs to	The size in bytes	The date of last modification	The name of the file

-rwxr-xr-x 1 bodryx bodryx 866 Feb 25 06:47 .bash_history

(R = read, W = write, X = execute, - no permission)

- - - normal file
- d - directory
- l - symbolic link
- p - named pipe
- s - socket
- b - block device
- c - character device

Users and Permissions - File and Directory

Permission	File	Directory
read	User can look at the contents of the file	User can list the files in the directory
write	User can modify the contents of the file	User can create new files and remove existing files in the directory
execute	User can use the filename as a UNIX command	User can change into the directory, but cannot list the files unless he or she has read permission. User can read files if he or she has read permission on them

Users and Permissions - Changing Permissions - Change Mode - Syntax

Tight control over file access is a major strength of Unix. So how do you change the permissions of your files?

$ chmod <mode> <file>

<mode> == [ugoa][+-=][rwx]

- Changes file/directory permissions based on **<mode>**
- The format for **<mode>** is a combination of 3 fields:
  - Who is a affected (any combination of **u**, **g**, **o**, **or a**)
  - Whether adding or removing permissions (**+** or **-** or **=**)
  - Which permissions are being **added/removed** (any combination of **r, w, x**)

Example: **ug+rx** - adds read and execute permissions for user and group

o-w - removes write permissions for others (no public writing)

Users and Permissions - Changing Permissions - Change Mode

Permissions may be specified as a sequence of 3 octal digits (octal is like decimal except that the digit range is 0 to 7 instead of 0 to 9). Each octal digit represents the access permissions for the user/owner, group and others respectively. The mappings of permissions onto their corresponding octal digits is as follows:

---	0
--x	1
-w-	2
-wx	3
r--	4
r-x	5
rw-	6
rwx	7

Users and Permissions - Changing Permissions - Example #1

$ chmod 600 music.txt

- sets the permissions on **music.txt** to **rw-------**
  - only the owner can read and write to the file

Users and Permissions - Changing Permissions - Example #2

Permissions may be specified symbolically, using the symbols **u (user)**, **g (group)**, **o(other)**, **a (all)**, **r (read)**, **w (write)**, **x (execute)**, **+ (add permission)**, **- (take away permission)** and **= (assign permission)**

$ chmod ug=rw,o-rw,a-x *.txt

- sets the permissions on all files ending in ***.txt** to **rw-rw----** (i.e. the owner and users in the file's group can read and write to the file, while the general public do not have any sort of access)
- **chmod** also supports a **-R** option **$ chmod -R go+r public**
  - will grant group and other read rights to the directory play and all of the files and directories within public

Users and Permissions - umask

umask (user mask) is a command and a function in POSIX environments that sets the file mode creation mask of the current process which limits the permission modes for files and directories created by the process. A process may change the file mode creation mask with umask and the new value is inherited by child processes.

- with umask you can define the permissions of the new files that your process will create

The user mask contains the octal values of the permissions you want to set for all the new files and to calculate the value of the umask **subtract the value of the permissions you want to get from 666 (for a file) or 777 (for a directory)**

For example, suppose you want to change the default mode for files to 664 (rw-rw-r–). The difference between 666 and 664 is 002, which is the value you would use as an argument to the umask command.

Users and Permissions - umask parameters

umask Octal Value	File Permissions	Directory Permissions
0	rw-	rwx
1	rw-	rw-
2	r--	r-x
3	r--	r--
4	-w-	-wx
5	-w-	-w-
6	--x	--x
7	--- (none)	--- (none)

Users and Permissions - umask - Examples

Example:

```
$ touch newfile
$ ls -l newfile
-rw-rw-r-- 1 username username 0 Feb 27 04:24 newfile
$ umask
0002
```

Example:

```
$ umask 0022
$ touch newfile2
ls -l newfile2
-rw-r--r-- 1 username username 0 Feb 27 04:30 newfile2
$ umask
0022
```

Commands - Basic Syntax

- Bash is the default shell
- Tokens are separated by whitespace
- Shell expects the first token to be a command
- All subsequent tokens are arguments
- Arguments that start with a dash, -, or two dashes, are called options
    - Used to modify the behavior of the command
- Non-option arguments are data passed to the command

Commands - Syntax Example

ls -a -l bodryx/docs

- ls – utility, to list contents of a directory
- -a – option, to include hidden files (all)
- -l – option, spit out long listing
- bodryx/docs – argument, directory to list

Short options can generally be stacked:

ls -al bodryx/docs

Commands - Getting Help - man and info - Your Most Important Commands

You just saw a new command - how do you figure out what it does?

The **man** Command syntax:

man <command name>

info <command name>

- Brings up the manual page (manpage) for the selected command
- Unlike Google results, manpages are system-specic
- Gives a pretty comprehensive list of all possible options/parameters
- Use /<keyword> to perform a keyword search in a manpage
- The n-key jumps to successive search results

Commands - Traversing the Filesystem

ls – lists file or contents of a directory (current directory by default)

- -a – show hidden files (all)
- -o, -l – long (and longer) listing
- -d – directory (don’t list out the contents)
- -F – Decorate names depending on filetype

pwd – print the working (current) directory

- prints the full path to the current directory
- handy on minimalist systems when you get lost

cd [dirname] – change directories to [dirname]

- by default, takes you to the current user's home directory
- same command used in DOS
- can be given either an absolute path or a relative path to the destination directory

Commands - Directories - Create and Remove

mkdir [options] <directory>

- Makes a new directory with the specified name "**directory**"
- Can use relative/absolute paths to make directories outside the current one

By default, **rm** can't remove directories - we have a special command for that

rmdir [options] <directory>

- Removes an empty directory. Safe. it won't remove non-empty directories
- Throws an error if the directory is not empty
- The opposite of **mkdir**
- To delete a directory with all of its subdirectories and file contents, use **rm -r <directory>**
    - Directories can be moved/renamed using **mv**
    - Entire directories can be copied using **cp -r**

Commands - Creating a New File

The simplest way to create an empty le is by using the touch command

Syntax of touch:

touch [options] <file>

- Adjusts the timestamp of the specified file
- With no options, uses the current date/time
- More importantly, if the le doesn't exist, touch creates it

File extensions (**.exe, .txt, etc**) often don't matter in UNIX. Using touch to create a file results in a blank plain-text file - you don't need to add ".txt" to use it

Commands - Removing files

Removing files is at least as important as creating them, but it's a lot more dangerous too - there is no easy way to undo a file deletion

rm [options] <filename> – remove file

- -r – recursive. Careful, here
- -f – force. Ignore nonexistent files
- Using wildcards allows you to remove multiple files with a single command
  - **rm *** - Removes every file in the current directory
  - **rm *.jpg** - Removes every .jpg file in the directory
  - **rm *7*** - Removes every file with a 7 in its name

Commands - Copying

cp [options] <file> <destination> – copies a file from one location to another

- -i – interactive. Ask before overwriting destination file (if it exists)
- to copy multiple files, use the asterisk wildcard (*)
- to copy a complete directory, use **cp -r <src> <dest>**

Example:

$ cp -r /home/bodryx/* allfiles/ - copies all files and directories from the directory **/home/bodryx/** to **/home/<username>/allfiles/**

Commands - Moving, Renaming

mv [options] <source> <destination>

- move or rename, you can give the file a different name as you move it
  - Moves a file or directory from one place to another
  - Also used for renaming - just move from <oldname> to <newname>
  - -i – interactive. Ask before overwriting destination file (if it exists)

Commands - Specifying multiple filenames

Multiple filenames can be specified using special pattern-matching characters
The rules are:

- **'?'** matches any single character in that position in the filename
- **'*'** matches zero or more characters in the filename
- A **'*'** on its own will match all files
- **'*.*'** matches all files with containing a '.'
- Characters enclosed in square brackets (**'['** and **']'**) will match any filename that has one of those characters in that position.
- A list of comma separated strings enclosed in curly braces (**"{"** and **"}"**) will be expanded as a Cartesian product with the surrounding characters

For example:

1. **???** matches all three-character filenames
2. **?ell?** matches any five-character filenames with **'ell'** in the middle
3. **he*** matches any filename beginning with '**he**'
4. **[m-z]*[a-l]** matches any filename that begins with a letter from **'m'** to **'z'** and ends in a letter from **'a'** to **'l'**
5. **{/usr,}{/bin,/lib}/file** expands to **/usr/bin/file**, **/usr/lib/file**, **/bin/file**, and **/lib/file**

Commands - Making Hard and Soft (Symbolic) Links

Direct (hard) and indirect (soft or symbolic) links from one file or directory to another can be created using the **ln** command.

$ ln filename linkname

- creates another directory entry for filename called linkname (**linkname is a hard link**).

Both directory entries appear identical (and both now have a link count of 2). If either filename or linkname is modified, the change will be reflected in the other file (since they are in fact just two different directory entries pointing to the same file).

$ ln -s filename linkname

- creates a shortcut called linkname (**linkname is a soft link**). The shortcut appears as an entry with a special type ('l')

Example:

$ ln -s /home/bodryx/README softlink
$ ls -l softlink
lrwxrwxrwx 1 username username 19 Feb 27 03:01 softlink -> /home/bodryx/README

Commands - Archiving

tar – “tape archive and retrieval” combines multiple files into one

- An archive is a file that contains other files plus information about them, such as their filename, owner, timestamps, and access permissions. **tar** does not perform any compression by default

Example:

$ tar -cvzf apache-tomcat.tgz apache-tomcat

- -c - create a new archive
- -z, - create gzip archive
- -f - file

$ tar -xvzf apache-tomcat.tar.gz (you can use *.tar.gz or *.tgz)

- -x, - extract
- -v, - verbosely list files processed

Commands - Filters

Programs that read some input, perform some transformation, write out the results

- **head, tail** – Displays first (last) n lines of input
- **find, which, locate -** To find file
- **grep** – Search input using regular expressions
- **sort** – Sorts input by lines (lexically, or numerically)
- **uniq** – Unique, removes identical, adjacent lines
- **wc** – Word count (line count, character count)

Commands - Filters - head and tail

head and **tail** display the first and last few lines in a file respectively. You can specify the number of lines as an option.

Example:

$ head -10 /var/log/auth.log

$ tail -25 /var/log/auth.log

tail includes a useful **-f** option that can be used to continuously monitor the last few lines of a (possibly changing) file. This can be used to monitor log files, for

Example:

$ tail -f /var/log/auth.log - continuously outputs the latest additions to the system log file

Commands - Filters - Finding Files - find

find directory -name targetfile -print

find will look for a file called **targetfile** in any part of the directory tree rooted at directory. **targetfile** can include wildcard characters.

For example:

$ find /home -name "*.txt" -print 2>/dev/null

- will search all user directories for any file ending in "**.txt**" and output any matching files (with a full absolute or relative path). Here the quotes (") are necessary to avoid filename expansion, while the **2>/dev/null** suppresses error messages (arising from errors such as not being able to read the contents of directories for which the user does not have the right permissions)

Commands - Filters - Finding Files - which

$ which (sometimes also called whence) command

If you can execute an application program or system utility by typing its name at the shell prompt, you can use which to find out where it is stored on disk.

Example:

$ which ls

/bin/ls

Commands - Filters - Finding Files - locate

$ locate string

find can take a long time to execute if you are searching a large filespace.

The **locate** command provides a much faster way of locating all files whose names match a particular search string.

Example:

$ locate ".txt" - will find all filenames in the filesystem that contain ".txt" anywhere in their full paths.

(One disadvantage of **locate** is it stores all filenames on the system in an index that is usually updated only once a day. This means **locate** will not find files that have been created very recently. It may also report filenames as being present even though the file has just been deleted. Unlike **find**, **locate** cannot track down files on the basis of their **permissions**, **size** and so on.)

Commands - Filters - grep - (General Regular Expression Print)

$ grep options pattern files

- grep searches the named files (or standard input if no files are named) for lines that match a given pattern. The default behaviour of grep is to print out the matching lines

Example:

$ grep find /home/bodryx/*

- searches all files in the directory **/home/bodryx/** for lines containing "find".

Commands - Filters - grep

Some of the more useful options that grep provides are:

- **-c** (print a count of the number of lines that match)
- **-i** (ignore case)
- **-v** (print out the lines that don't match the pattern)
- **-n** (print out the line number before printing the matching line)

Example:

$ grep -vi find /home/bodryx/*

- searches all files in the directory **/home/bodryx/** for lines that do not contain any form of the word find (e.g. Find, FIND, or fINd).

Commands - Filters - grep

If you want to search all files in an entire directory tree for a particular pattern, you can combine grep with find using backward single quotes to pass the output from find into grep.

Example:

```
$ grep error `find . -name "*" -print`
```

- will search all text files in the directory tree rooted at the current directory for lines containing the word "error".

Commands - Filters - sort

There are two facilities that are useful for sorting files in UNIX:

$ sort filenames

- **sort** sorts lines contained in a group of files alphabetically (or if the -n option is specified) numerically. The sorted output is displayed on the screen, and may be stored in another file by redirecting the output

Example:

$ sort /home/bodryx/input1.txt /home/bodryx/input2.txt > output.txt

- outputs the sorted concatenation of files **input1.txt** and **input2.txt** to the file **output.txt**

Commands - Filters - uniq

$ uniq filename

- uniq removes duplicate adjacent lines from a file. This facility is most useful when combined with sort

Example:

$ sort output.txt | uniq > output1.txt

Commands - Some Other Utilities

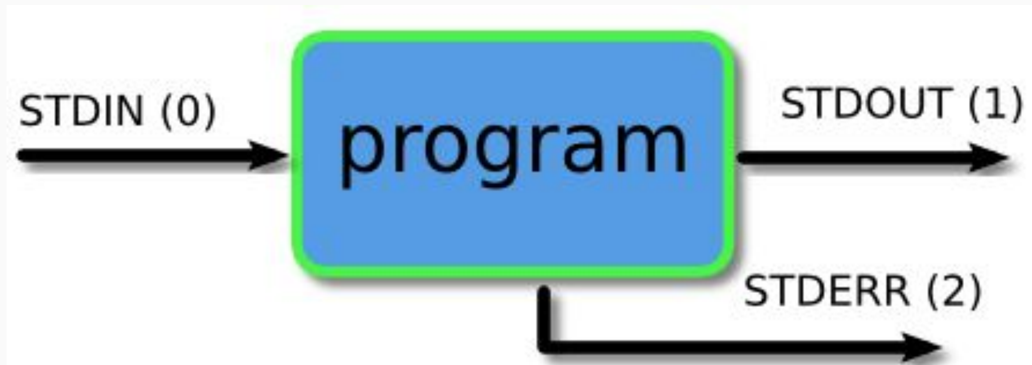
- **cal** – Print calendar
- **date** – Print current date and time
- **time** – Does not show you the current time
- **chown** – Change the user and group ownership of a file
- **who** – Print who is currently logged in
- **w** – Displays a list of the logged users like who, but also display their attached process and the uptime of the machine you’re on
- **finger user** – more information about user
- **du -sh** – Disk usage summary

Commands - Piping and Redirection

Every program we run on the command line automatically has three data streams connected to it.

- **STDIN** (0) - Standard input (data fed into the program)
- **STDOUT** (1) - Standard output (data printed by the program, defaults to the terminal)
- **STDERR** (2) - Standard error (for error messages, also defaults to the terminal)

Piping and redirection is the means by which we may connect these streams between programs and files to direct data in interesting and useful ways.



Commands - Redirecting to a File

Operator (**>**) indicates to the command line that we wish the programs output (or whatever it sends to **STDOUT**) to be saved in a file instead of printed to the screen.

Example:

```
$ ls /home/bodryx/
input1.txt  input2.txt  public  README
$ ls /home/bodryx/ > myoutput
$ cat myoutput
input1.txt
input2.txt
public
README
```

If we redirect to a file which does not exist, it will be created automatically for us. If we save into a file which already exists, however, then it's contents will be cleared, then the new output saved to it.

Commands - Redirecting to a File - To append

We can instead get the new data to be appended to the file by using the double greater than operator (**>>**)

Example:

```
$ ls /home/bodryx/ >> myoutput
$ cat myoutput
input1.txt
input2.txt
public
README
input1.txt
input2.txt
public
README
```

Commands - Redirecting from a File

If we use the less than operator (**<**) then we can send data the other way. We will read data from the file and feed it into the program via it's **STDIN** stream.

Example:

```
$ wc -l myoutput
8 myoutput
$ wc -l < myoutput
8
```

We may easily combine the two forms of redirection we have seen so far into a single command as seen in the example below.

Example:

```
$ wc -l < myoutput > myoutputnew
$ cat myoutputnew
8
```

Commands - Redirecting STDERR

STDERR is stream number **2** and we may use these numbers to identify the streams. If we place a number before the **>** operator then it will redirect that stream (if we don't use a number, like we have been doing so far, then it defaults to stream **1**).

Example:

```
$ ls -l /home/bodryx/public work
ls: cannot access 'work': No such file or directory
/home/bodryx/public:
total 0
$ ls -l /home/bodryx/public work 2> errors.txt
/home/bodryx/public:
total 0
$ cat errors.txt
ls: cannot access 'work': No such file or directory
```

Commands - Piping

Piping is mechanism for sending data from one program to another and the operator we use is (|). What this operator does is feed the output from the program on the left as input to the program on the right.

Example:

```
$ ls /home/bodryx/
input1.txt  input2.txt  public  README
$ ls /home/bodryx/ | head -3
input1.txt
input2.txt
public
```

We may pipe as many programs together as we like. In the below example we have then piped the output to tail so as to get only the third file.

Example:

```
$ ls /home/bodryx/ | head -3 | tail -1 > myoutput
$ cat myoutput
public
```