
EE324 – Applications Programming for Embedded Systems

Lecture 01(b) Review of Unix Basics

1.1 - Introduction to Unix

1.2 - Unix Shells

1.3 - Basic Unix Commands

1.4 - Unix Editors

1.5 - Files & Directories

1.6 - Security & Permissions



Files and Directories



Files and Directories (1)

◆ What is a file?

- a container for ordered data
- persistent (stays around) and accessible by name

◆ Unix files

- regular Unix files are pretty simple
 - ❖ essentially a sequence of bytes
 - ❖ can access these bytes in order, or access a particular offset from the front
- Unix files are identified by a name in a directory
 - ❖ this name is actually used to resolve the
 - hard disk name/number, the cylinder number, the track number, the sector, the block number
 - you see none of this
 - ❖ it allows the file to be accessed

Files and Directories (2)

- ◆ Unix files come in other flavours as well
 - Directories
 - ❖ a file containing pointers to other files
 - ❖ equivalent of a “folder” on a Mac or Windows
 - Links
 - ❖ a pointer to another file
 - ❖ used like the file it points to
 - ❖ similar to “shortcuts” in Windows, but better
 - Devices
 - ❖ access a device (like a soundcard, or mouse, or ...) like it is a file
 - Etc.

Directories (1)

- ◆ Current Working Directory
 - the directory you are looking at right now
 - the shell remembers this for you
- ◆ To determine the Current Working Directory, use the command **pwd** (Print Working Directory)

Use: **obelix[18] > pwd**

Result: print the current working directory

E.g. **/csd/faculty/katchab/**

Directories (2)

◆ Moving about the filesystem

- Use the “**cd**” (Change Directory) command to move between directories and change the current directory

Use: **obelix[19] > cd 211**

Result: Makes **cs211** the current working directory

◆ Listing the contents of a directory

- Use the “**ls**” (LiSt directory) command to list the contents of a directory

obelix[20] > ls

tmp/ a.out* smit.script cs211@

```
graph TD; D[Directories] --> tmp[tmp/]; E[Executable] --> aout[a.out*]; L[Link] --> cs211[cs211@];
```

Directories (3)

◆ The upside-down tree

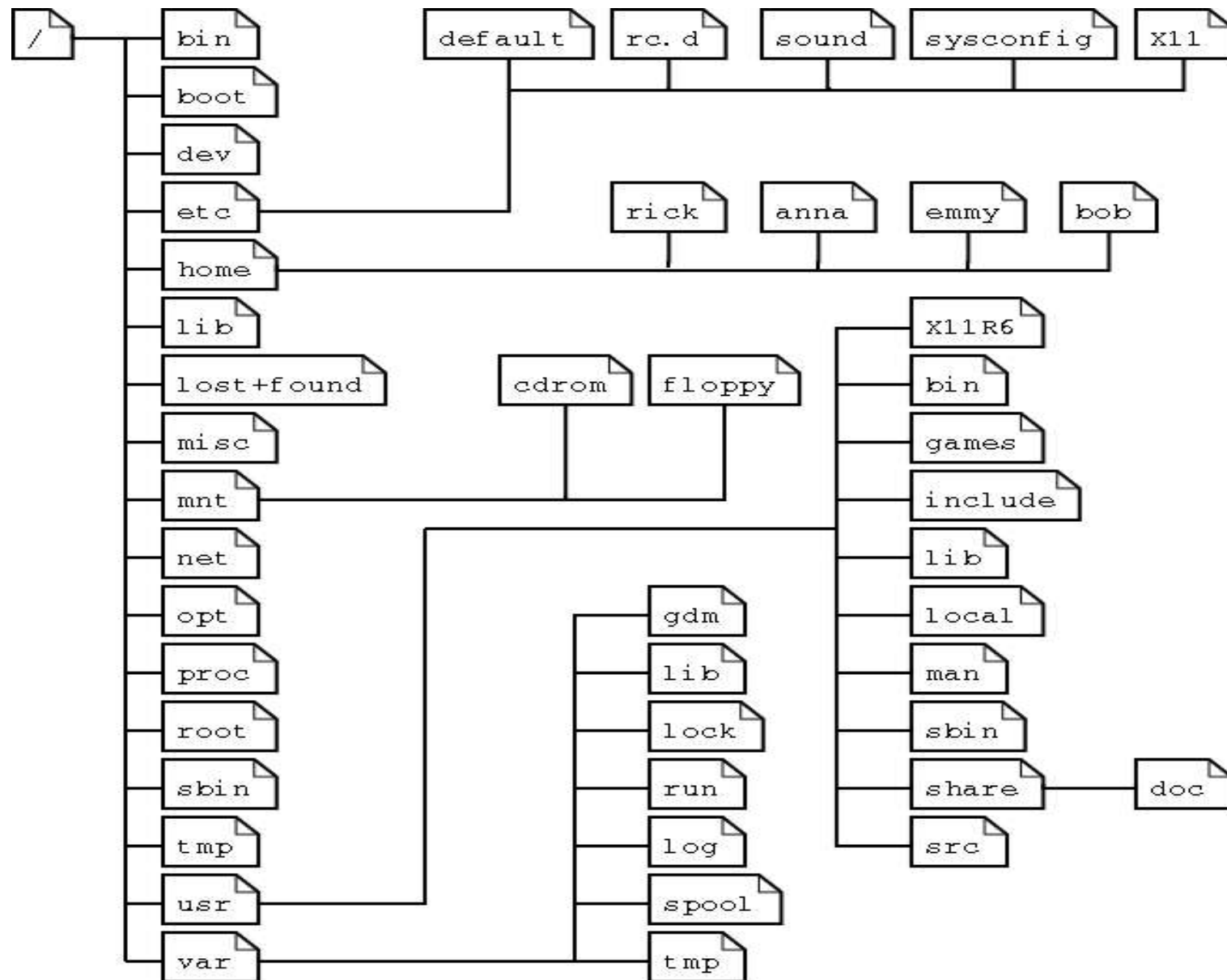
- the Unix filesystem is organized like an upside-down tree
 - ❖ at the top of the filesystem is the root
 - write this as a lone slash: `/`
 - this is NOT a backslash (opposite of MS-DOS)!
 - ❖ For example, you can change to the root directory:

```
obelix[21] > cd /
```

```
obelix[22] > ls
```

```
TT_DB/  dev/    home/    mnt/    sbin/  xfn/  
bin@    devices/ kernel/  net/    tmp/   cdrom/  
etc/    lib@    opt/     usr/    core  export/  
local/  platform/ var/     courses@ lost+found/  
proc/   vol/
```

Root Directory



Subdirectories of the root directory (1)

- **/bin** Common programs, shared by the system, the system administrator and the users.
- **/boot** The startup files and the kernel, vmlinuz. In recent distributions also Grub data. Grub is the GRand Unified Boot loader and is an attempt to replace the many different boot-loaders we know today.
- **/dev** Contains references to all the CPU peripheral hardware, which are represented as files with special properties.
- **/etc** Most important system configuration files are in /etc, this directory contains data similar to those in the Control Panel in Windows
- **/home** Home directories of the common users.
- **/initrd** (on some distributions) Information for booting. Do not remove!
- **/lib** Library files, includes files for all kinds of programs needed by the system and the users.

Subdirectories of the root directory (2)

- **/lost+found** Every partition has a lost+found in its upper directory. Files that were saved during failures are here.
- **/misc** For miscellaneous purposes.
- **/mnt** Standard mount point for external file systems, e.g. a CD-ROM or a digital camera.
- **/net** Standard mount point for entire remote file systems
- **/opt** Typically contains extra and third party software.
- **/proc** A virtual file system containing information about system resources. More information about the meaning of the files in proc is obtained by entering the command `man proc` in a terminal window. The file `proc.txt` discusses the virtual file system in detail.
- **/root** The administrative user's home directory. Mind the difference between `/`, the root directory and `/root`, the home directory of the root user.

Subdirectories of the root directory (3)

- **/sbin** Programs for use by the system and the system administrator.
- **/tmp** Temporary space for use by the system, cleaned upon reboot, so don't use this for saving any work!
- **/usr** Programs, libraries, documentation etc. for all user-related programs.
- **/var** Storage for all variable files and temporary files created by users, such as log files, the mail queue, the print spooler area, space for temporary storage of files downloaded from the Internet, or to keep an image of a CD before burning it.

❖ How can you find out which partition a directory is on?
Using the **df** command with a dot (.) as an option shows the partition the current directory belongs to, and informs about the amount of space used on this partition:

Pathnames (1)

- ◆ A typical Unix file system spans many disks
 - As a user you don't know or need to know which physical disk things are on
 - ❖ in fact, you don't even know which machine they are attached to: disks can be “remote”
 - Inside each directory may be more directories
- ◆ The Absolute Path
 - to identify where a file is, string the directories together
 - ❖ separating names with slashes:
 - ❖ e.g. `/home/student/1991/katleen`
 - ❖ this is the absolute path for Katleen's home directory
 - ❖ lists everything from the root down to the directory you want to specify

Pathnames (2)

- ◆ When you first log in, you are in your HOME directory
 - To see what this is:
`obelix[1] > pwd`
`/home/student/1991/katleen`
 - Your home directory is also stored in the environment variable HOME
`obelix[2] > echo Katleen's home is $HOME`
`Katleen's home is /home/student/1991/katleen`
 - You can “Go Home” by typing
`obelix[3] > cd $HOME`

Pathnames (3)

◆ Some shorthand

- In some shells (including tcsh, csh, and bash), \$HOME can be abbreviated as ~ (tilde)
- Example: obelix[26] > cd ~/bin
 - ❖ change to the bin directory under your home directory (equivalent to \$HOME/bin)
 - ❖ this is where you usually store your own commands or “executables”
- To quickly go home:
obelix[27]% cd
with no parameters, cd changes to your home directory
- ~user refers to the home directory of user
 - ❖ For me, ~katleen is the same as ~
 - ❖ ~pcor refers to Peter Corcoran's home directory (/home/student/1991/pcorcoran)

Pathnames (4)

◆ Relative pathnames

- You can also specify pathnames relative to the **current** working directory
 - ❖ This is called a **relative pathname**
- For example

```
obelix[28] > pwd
```

```
/home/student/1991/katleen
```

```
obelix[29] > ls
```

```
tmp/      a.out*      smit.script  ee324@
```

```
obelix[30] > cd tmp
```

```
obelix[31] > pwd
```

```
/gaul/s1/student/1991/katchab/tmp
```

- ❖ Note: You don't need to know absolute pathnames

- ◆ For most commands which require a file name, you can specify a pathname (relative or absolute)

Pathnames (5)

- ◆ Every directory contains two “special” directories: “.” and “..”

“.” : another name for the current directory

❖ e.g. `cp cs211/foo .`

“..” : another name for the immediate parent directory of the current directory

- ◆ Use this to cd to your parent:

```
obelix[32] > pwd  
/home/student/1991/katleen
```

```
obelix[33] > cd ..  
obelix[34] > pwd  
/home/s1/student/1991
```

```
obelix[35] > cd ../..  
obelix[36] > pwd  
/home/s1
```



Pathnames (6)

- ◆ You can locate a file or directory by this way:
 - look at the first character of the pathname
 - ❖ / start from the root
 - ❖ . start from the current directory
 - ❖ .. start from the parent directory
 - ❖ ~ start from a home directory
 - ❖ else start from the current directory
 - going down to the subdirectories in the pathname, until you complete the whole pathname.
 - for example:
 - ❖ /home/s1/student/1991/katleen/ee324/readme.txt
 - ❖ ~/ee324/readme.txt
 - ❖ ee324/readme.txt

Working with Directories (1)

- ◆ Create a directory with the **mkdir** command
mkdir newdirname
- ◆ newdirname can be given with pathname

```
obelix[37] > pwd
/home/s1/student/1991/katleen/ee324/
obelix[38] > ls
readme.txt
obelix[39] > mkdir mydir1
obelix[40] > ls
readme.txt  mydir1/
obelix[41] > mkdir mydir1/mydir2
obelix[42] > ls mydir1
mydir2/
obelix[43] > cd mydir1/mydir2
```



Note: we can specify
a directory with ls

Working with Directories (2)

◆ Remove a directory with the `rmdir` command


`rmdir dirname`

- `dirname` is the directory to remove and can be specified using a pathname
- if the directory exists and is empty it will be removed

◆ Examples:

```
obelix[44] > cd ~/cs211; ls  
readme.txt mydir1/  
obelix[45] > ls mydir1  
mydir2/
```

Assuming mydir1/mydir2
is still empty



```
obelix[46] > rmdir mydir1/mydir2  
obelix[47] > ls mydir1  
obelix[48] > rmdir mydir1
```

mydir1 is now empty,
so this will work fine



Working with Directories (3)

- ◆ Move a file from one directory to another

```
obelix[1] > pwd
```

```
/home/s1/student/1991/katleen/ee324/
```

```
obelix[2] > ls
```

```
readme.txt mydir1/
```

```
obelix[3] > ls mydir1
```

```
hello.txt
```

```
obelix[4] > mv mydir1/hello.txt .
```

A dot is here.



```
obelix[5] > ls mydir1
```

```
obelix[6] > ls
```

```
readme.txt hello.txt mydir1/
```

- ◆ You can also move a directory the same way - it is just a special file, after all.

Working with Directories (4)

◆ Copy a file from one directory to another

```
obelix[1] > ls
```

```
readme.txt mydir1/
```

```
obelix[2] > cp readme.txt mydir1
```

```
obelix[3] > ls mydir1
```

```
readme.txt
```

◆ Copying a directory

```
obelix[4] > cp mydir1 mydir2
```

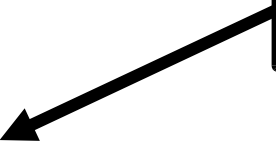
```
cp: mydir1: is a directory
```

```
obelix[5] > cp -r mydir1 mydir2
```

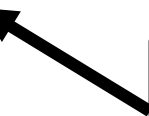
```
obelix[6] > ls mydir2
```

```
readme.txt
```

Cannot use just cp
to copy a directory

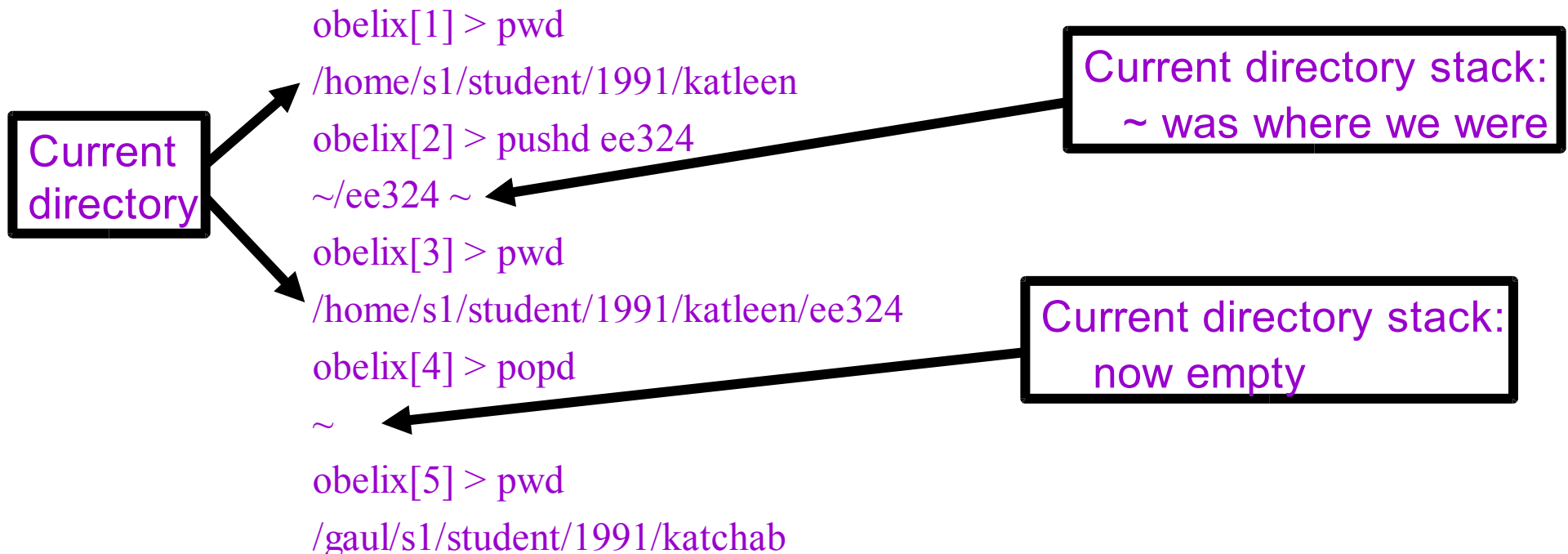


Must do a recursive copy
(cp -r) to copy a directory



Working with Directories (5)

- ◆ Some shells (csh and tcsh) provide **pushd** and **popd** directory commands
- ◆ **pushd** changes directories, but remembers the previous one by pushing it on to a stack
- ◆ **popd** changes directories back to the last directory placed on the stack by **pushd**



Working with Directories (6)

- ◆ What if you need to locate a file, or set of files, in a large directory structure?
 - Using `cd` and `ls` would be very tedious!
- ◆ The command `find` is used to search through directories to locate files.
 - Wildcards can be used, if the exact file name is unknown, or to find multiple files at once.
 - Can also find files based on size, owner, creation time, type, permissions, and so on.
 - Can also automatically execute commands on each file found.
- ◆ Do a “`man find`” for details and examples!

More Files and Directories (1)

◆ What files do I already have?

- Startup files for bash and tcsh (.login, .bashrc)
- Contain commands run after you type your password, but before you get a prompt
- Assume you've not used your account before

```
obelix[1] > ls
```

```
obelix[2] >
```

- Why can't I see any files?
 - ❖ Files beginning with a 'dot' are usually control files in Unix and not generally displayed
- Use the -a option to see all files

```
obelix[3] > ls -a
```

```
./  ../  .cshrc .login
```

```
obelix[4] >
```


Unix Filenames (1)

- ◆ Almost any character is valid in a file name
 - all the punctuation and digits
 - the one exception is the / (slash) character
 - the following are not encouraged
 - ❖ ? * [] “ ” ’ () & : ; !
 - the following are not encouraged as the first character
 - ❖ - ~
 - control characters are also allowed, but are not encouraged
- ◆ UPPER and lower case letters are different
 - A.txt and a.txt are different files

Unix Filenames (2)

- ◆ No enforced extensions

- The following are all legal Unix file names

- ❖ a

- ❖ a.

- ❖ .a

- ❖ ...

- ❖ a.b.c

- ◆ Remember files beginning with dot are hidden

- `ls` cannot see them, use `ls -a`

- ◆ `.` and `..` are reserved for current and parent directories

Unix Filenames (3)

- ◆ Even though Unix doesn't enforce extensions,
 - “.” and an extension are still used for clarity
 - ❖ .jpg for JPEG images
 - ❖ .tex for LaTeX files
 - ❖ .sh for shell scripts
 - ❖ .txt for text files
 - ❖ .mp3 for MP3's
 - some applications may enforce their own extensions
 - ❖ Compilers look for these extensions by default
 - .c means a C program file
 - .C or .cpp or .cc for C++ program files
 - .h for C or C++ header files
 - .o means an object file

Unix Filenames (4)

- ◆ Executable files usually have no extensions
 - cannot execute file `a.exe` by just typing `a`
 - telling executable files from data files can be difficult
- ◆ “`file`” command
 - Use: `file filename`
 - Result: print the type of the file
 - Example: `obelix[1] > file ~/.cshrc`
`.cshrc: executable c-shell script`
- ◆ Filenames and pathnames have limits on lengths
 - 1024 characters typically
 - these are pretty long (much better than MS-DOS days and the 8.3 filenames)

Filename Wildcarding (1)

- ◆ Wildcarding is the use of “special” characters to represent or match a sequence of other characters
 - a short sequence of characters can match a long one
 - a sequence may also match a large number of sequences
- ◆ Often use wildcard characters to match filenames
 - filename substitution – generally known as “**globbing**”
- ◆ Wildcard characters
 - * matches a sequence of zero or more characters
 - Example: **a*.c*** matches abc.c, abra.cpp,
 - ? matches any single character
 - Example: **a?.c** matches ab.c, ax.c, but not abc.c
 - [...] matches any character between the braces
 - Example: **b[aei]t** matches bat, bet, or bit, not baet

Filename Wildcarding (2)

- ◆ Wildcard sequences can be combined

obelix[6] > mv a*.[ch] cfiles/

- ❖ mv all files beginning with a and ending with .c or .h into the directory cfiles

obelix[7] > ls [abc]*.?

- ❖ list files whose name begins with a, b, or c and ends with . (dot) followed by a single character

- ◆ Wildcards do not cross "/" boundaries

- Example: katleen*c does not match katleen/codec

- ◆ Wildcards are expanded by the shell, and not by the program

- Programmers of commands do not worry about searching the directory tree for matching file names
- The program just sees the list of files matched

Filename Wildcarding (3)

◆ Matching the dot

- A dot (.) at
 - ❖ the beginning of a filename, or
 - ❖ immediately following a /must be matched explicitly.
- Similar to the character /
- Example:

`obelix[8] > cat .c*`

cat all files whose names
begin with .c

◆ As mentioned earlier, [...] matches any one of the characters enclosed

- Within “[...]”, a pair of characters separated by “-” matches any character lexically between the two
 - ❖ Example:

`obelix[9] > ls [a-z]*`

lists all files beginning
with a character between
ASCII 'a' and ASCII 'z'

Filename Wildcarding (4)

- ◆ More advanced examples:

- What does the following do?

```
obelix[10] > ls /bin/*[-_]*
```

- What about this?

```
obelix[11] > ls *
```

- What about this?

```
obelix[12] > mv *.bat *.bit
```


Unix Quoting (1)

◆ Double Quotes: "..."

- Putting text in double quotes "..." stops interpretation of some shell special characters (whitespace mostly)
- Examples:

```
obelix[12] > echo Here are some words
```

```
Here are some words
```

```
obelix[13] > echo "Here are some words"
```

```
Here are some words
```

```
obelix[14] > mkdir "A directory name with spaces! "
```

```
obelix[15] > ls A*
```

```
A directory name with spaces!/  
A directory name with spaces!
```

Unix Quoting (2)

◆ Single Quotes '...'

- Stops interpretation of even more specials

- ❖ Stop variable expansion (\$HOME, etc.)

- ❖ Backquotes `...` (execute a command and return result ...we'll get to this later)

- ❖ Note difference: single quote ('), backquote (`)

- ❖ Examples:

```
obelix[16] > echo "Welcome $HOME"
```

```
Welcome /gaul/s1/student/1991/katchab
```

```
obelix[17] > echo 'Welcome $HOME'
```

```
Welcome $HOME
```

Unix Quoting (3)

◆ Backslash \

- ‘quotes’ the next character
- Lets one escape all of the shell special characters

```
obelix[18] > mkdir Dir\ name\ with\ spaces\ *\ *
```

```
obelix[19] > ls Dir\ *
```

```
Dir name with spaces**/
```

- Use backslash to escape a newline character

```
obelix[20]% echo "This is a long line and\  
we want to continue on the next"
```

```
This is a long line and we want to continue on the next
```

- Use backslash to escape other shell special chars

- ❖ Like quote characters

```
obelix[21] > echo \"Bartlett's Familiar Quotations\  
\"Bartlett's Familiar Quotations"
```

Unix Quoting (4)

◆ Control-V

- Quotes the next character, even if it is a control character
- Lets one get weird stuff into the command line
- Very similar to backslash but generally for ASCII characters which do not show up on the screen
- Example: the backspace character

```
obelix[22] > echo "abc^H^H^Hcde"
```

```
cde
```

Control-h is backspace
on most terminals

typing Control-v Control-h
enters a "quoted" Control-h
to the shell
• written ^H

- Precisely how it works is dependant on the shell you use, and the type of terminal you are using

Hard and Symbolic Links (1)

- ◆ When a file is created, there is one link to it.
- ◆ Additional links can be added to a file using the command `ln`. These are called **hard links**.
- ◆ Each hard link acts like a pointer to the file and are indistinguishable from the original.

```
obelix[1] > ls
```

```
readme.txt
```

```
obelix[2] > ln readme.txt unix_is_easy
```

```
obelix[3] > ls
```

```
readme.txt  unix_is_easy
```

- ◆ There is only one copy of the file contents on the hard disk, but now two distinct names!

Hard and Symbolic Links (2)

- ◆ A symbolic link is an indirect pointer to another file or directory.
- ◆ It is a directory entry containing the pathname of the pointed to file.

```
obelix[1] > cd
```

```
obelix[2] > ln -s /usr/local/bin bin
```

```
obelix[3] > ls -l
```

```
lrwxrwxrwx bin -> /usr/local/bin
```

```
.....
```

```
obelix[4] > cd bin
```

```
obelix[5] > pwd
```

```
/usr/local/bin
```

Hard and Symbolic Links (3)

- ◆ Two hard links have the same authority to a file
 - Removing any of them will NOT remove the contents of the file
 - Removing all of the hard links will remove the contents of the file from the hard disk.
- ◆ A symbolic link is just an entry to the real name
 - Removing the symbolic link does not affect the file
 - Removing the original name will remove the contents of the file
- ◆ Only super users can create hard links for directories
- ◆ Hard links must point to files in the same Unix filesystem



Security and Permissions



File Permissions (1)

- ◆ With respect to a particular file, Unix divides the set of all users on a system into three categories:
 - user
 - ❖ The owner of the file.
 - group users
 - ❖ You are all probably in the group **2ndyr**.
 - ❖ Used for easier administration of access control.
 - ❖ Normally only the superuser can set up groups.
 - others (aka “World”)
 - ❖ Everyone else.

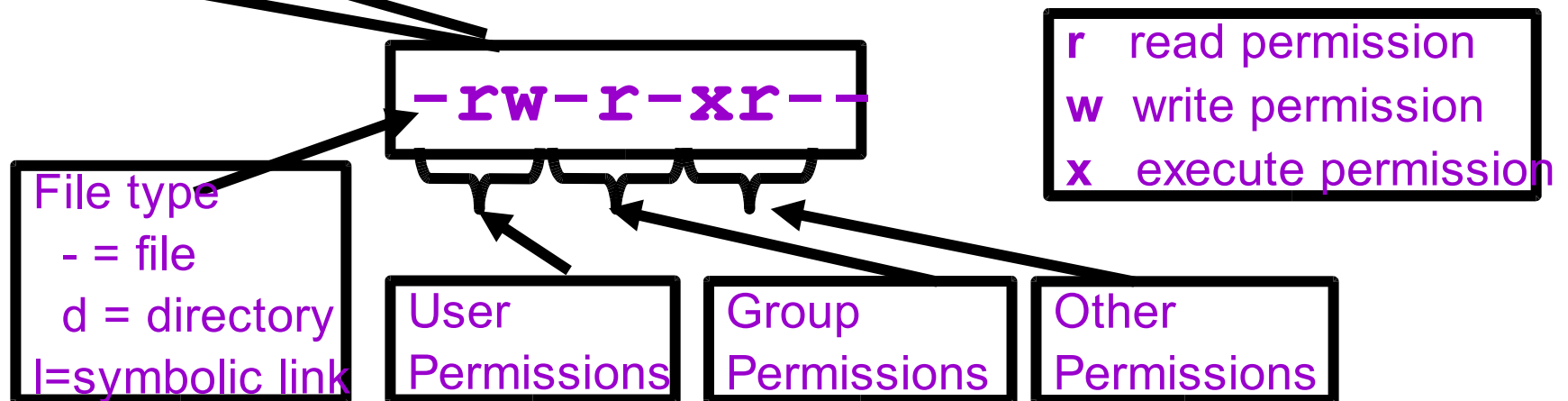
File Permissions (2)

- ◆ Permissions can be viewed with the `ls -l` command

```
obelix[1] > ls -l
```

```
total 1247
```

```
-rw-----    1 katleen  1117   Jul 23 15:49  bad.cpp
drwx--x--x    2 katleen  2048   Jul 17 10:13  bibd/
drwxr-xr-x    2 katleen   512   Aug 27 23:18  cache/
-rw-----    1 katleen  2081   Jul 23 15:49  tst2.s
-rw-r-xr--    1 katleen  1275   Jul 23 15:49  vecexpr.cpp
```



File Permissions (3)

- ◆ Permissions are changed with the **chmod** command.
- ◆ There are two syntaxes you can use:
chmod DDD file [file ...]
 - DDD are 3 octal digits representing bits of protection
 - rwx rwx rwx can be thought of as 111 111 111 in binary

rw-	r--	r--	
110	100	100	
6	4	4	chmod 644 file

File Permissions (4)

- ◆ `chmod [ugoa][+ -=][rwx] file [...]`
 - This is the “symbolic” method.
 - `chmod u+rwx file` gives the User Read, Write, and eXecute
 - `chmod g+rx file` gives the Group Read and eXecute
 - `chmod o-rwx file` removes R, W, and X from Others
 - `chmod a+x file` gives All eXecute permission
 - `chmod g=r file` gives Group Read permission and makes sure it has nothing else

- ◆ Symbolic modes can be appended with commas
 - `chmod u=rwx,g-w,o-rwx file` for instance

The umask command

- ◆ **umask** sets the default permissions for any file you will create
- ◆ Format is **backwards** to the **chmod** command
 - tells you which permissions will **NOT** be given
 - ❖ **umask 077** means don't let anyone but the User do anything with my files by default
- ◆ Generally set umask once in your .cshrc file and never set it again

Directory Permissions (1)

- ◆ Directory permissions are different from the file permissions
 - Requires **execute** permission to **access** files in the directory and its subdirectories
 - Requires **read** permission to **list the contents** of the directory (does not affect the subdirectory)
 - Requires **write** permission to **create files** in the directory (does not affect the subdirectory)

Directory Permissions (2)

```
obelix[1] > ls -l
```

```
drwx--x--- 2048 Jul 17 10:13 bibd/
```

```
obelix[2] > ls -l bibd
```

```
-r--r--rwx 173 Jul 17 10:13 readme
```

- ◆ Files in bibd/ are accessible to user
- ◆ Files in bibd/ are accessible by name (if you know the name) for group users
- ◆ Files in bibd/ and subdirectories are not accessible to others.

Directory Permissions (3)

- ◆ The `-R` option to `chmod` is useful when working with directories.
 - It recursively changes the mode for each `chmod` operand that is a directory.
 - All files and directories would receive those permissions.
 - `chmod -R a+rw dir` gives everyone read and write permission to each file under `dir` (not execute though!!!)
 - `chmod -R a+rwX dir` gives the executable access to allow people to actually access the files under `dir`
 - ❖ Makes all files executable though ...
 - `chmod -R a+rwX dir` gives the executable access only to those files already executable (programs, directories, ...)