CS6 Practical System Skills

Fall 2019 edition

Leonhard Spiegelberg
lspiegel@cs.brown.edu

04 Users and Permissions

CS6 Practical System Skills

Fall 2019

Leonhard Spiegelberg *lspiegel@cs.brown.edu*

04.01 Permissions

UNIX is a multi-user system.

How do you protect files from other users, the world?

How do you share files with other users?

How do you protect one from oneself?

04.01 Users

On a system you'll find different *logical users*:

root ⇒ the OS account which has unlimited rights

admin ⇒ one or more accounts which may perform certain actions with root privileges

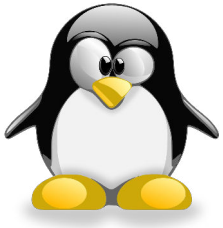
regular users ⇒ You, me & everyone other human out there

technical users ⇒ users created to run deployed programs with restricted privileges.

More on adding users, privileges, … in week 3 and the deployment lab.

How Unix categorizes users...

04.01 Users & Files



owner
creator of the file

group
multiple users

other
public, world

Each file is owned by a user ⇒ typically the creator

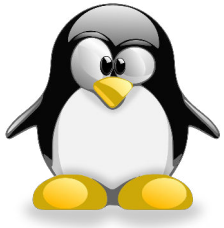
In addition, each file belongs to a group ⇒ smallest group: the user

04.01 Permissions for files

Each file in Unix has 3 permissions:

read	the file can be read, i.e. its contents displayed
write	the file can be modified or deleted
execute	the file can be run (i.e. executables or scripts)

04.01 Users and permissions



owner
creator of the file

group
multiple users

other
public, world

⇒ UNIX allows you to set (for each file) separate read/write/execute permissions for each party

04.01 Permissions for directories

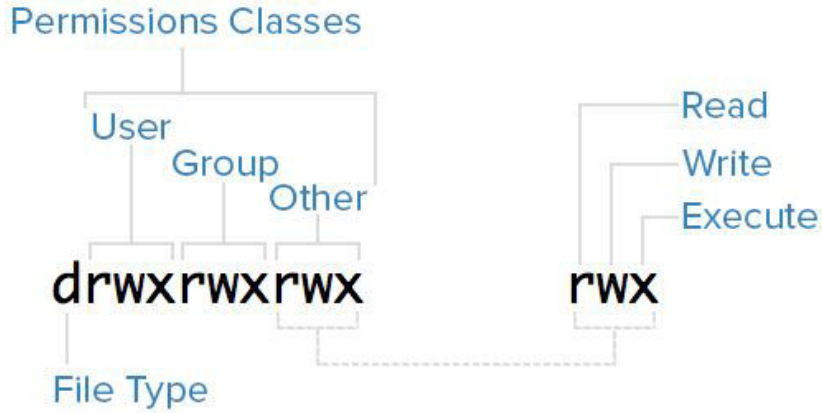
Because directories are also files, they have read, write, or execute permissions too. The meaning differs though:

permission	file	directory
read	Allows file to be read.	Allows file names in the directory to be read.
write	Allows file to be modified.	Allows entries to be modified within the directory.
execute	Allows file to be executed.	Allows access to contents and metadata for entries in the directory.

How can we get information about the permissions of a file?

04.02 ls -l ⇒ the longformat

owner and user are usually the same!
Terms are used interchangeably here often.

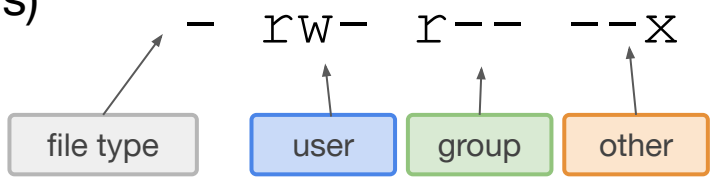



```
ls -l

total 88

-rw-r--r-- 1 sealion  friends      14  9 Sep 8:01 file.txt

-rw-r--r-- 1 sealion  friends   40390  9 Sep 9:00 penguin.jpg
```

permission string | owner | group

04.02 Permissions

permission string (10 characters)



filetype	symbol
regular file	-
directory	d
symbolic link	l
pipe	p
socket	s
block device	b
char device	c

permission	symbol
read	r
write	w
e**x**ecute	x

04.03 Setting permissions - chmod

`chmod mode file …`

change mode, i.e. set or update file permissions

⇒ only the owner (or root) can run this command for a file

⇒ mode can be either a number (numeric mode) or a combination of symbols

04.04 chmod - symbolic mode

party	symbol
user	u
group	g
other	o
all	a

action	symbol
add permission	+
remove permission	-
set to	=

permission	symbol
read	r
write	w
e**x**ecute	x

Example:

```
chmod u=rw,g=rx,o= file.txt
```

sets permissions for file.txt to -rw-r-x---

combine multiple statements with ,

04.05 chmod - numeric mode

Instead of using symbols, chmod can be used with an even short syntax using the following encoding.

Octal	Binary	String	Description
0	000	---	no permissions
1	001	--x	execute only
2	010	-w-	write only
3	011	-wx	write and execute
4	100	r--	read only
5	101	r-x	read and execute
6	110	rw-	read and write
7	111	rwx	read, write and execute

```
chmod u=rw,g=rx,o= file.txt ⇒ chmod 650 file.txt
```

04.05 chmod - numeric mode

⇒ combining permissions is adding numbers

4 = read 2 = write 1 = execute

Example: set user read and write permissions only:

	U	G	O
Symbolic	rw-	---	---
Binary	110	000	000
Decimal	6 = 4 + 2	0	0

```
⇒ chmod 600 file.txt
```

Octal	Binary	String	Description
0	000	---	no permissions
1	001	--x	execute only
2	010	-w-	write only
3	011	-wx	write and execute
4	100	r--	read only
5	101	r-x	read and execute
6	110	rw-	read and write
7	111	rwx	read, write and execute

04.06 ls -l revisited

Consider the following output from `ls -l`:

```
drwxr-xr-x 10 sealion animals 320 28 Nov 2018 lecture02
```

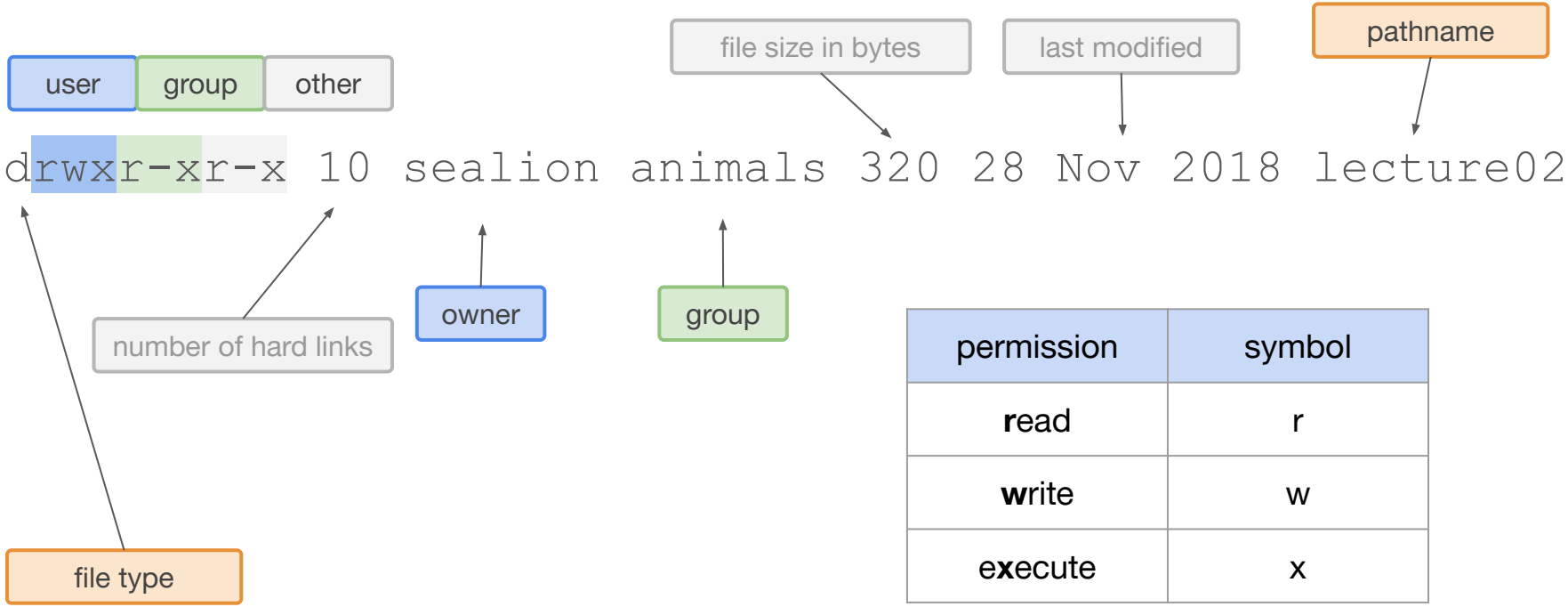
Who owns the file?

What permissions does lecture02 have?

What type of file is lecture02?

04.06 ls -l revisited

Consider the following output from `ls -l`:



permission	symbol
read	r
write	w
e**x**ecute	x

04.06 chmod - quiz

fill out the table, use ? if a permission bit can't be deducted.

file.txt permissions before	symbolic mode chmod	numeric mode chmod	file.txt permissions after
`rwx---rwx`	`chmod u=,g=,o= file.txt`		
`-w--w-rw-`		`chmod 777 file.txt`	
`-rw------`		`chmod 654 file.txt`	
	`chmod u=x`		`--xrwxrwx`
`---------`	`chmod u+r,u-r,u=rw`		

4 = read 2 = write 1 = execute

04.06 chmod - quiz

solutions:

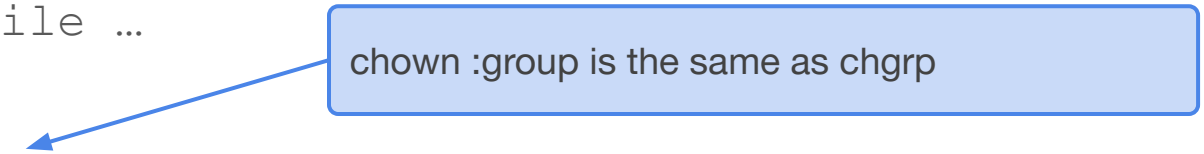
file.txt permissions before	symbolic mode chmod	numeric mode chmod	file.txt permissions after
rwx---rwx	chmod u=,g=,o= file.txt	chmod 000 file.txt	---------
-w--w-rw-	chmod u+rx,g=rwx,o+x file.txt	chmod 777 file.txt	rwxrwxrwx
-rw------	chmod u=rw,g=rx,o=r file.txt	chmod 654 file.txt	rw-r-xr--
???rwxrwx	chmod u=x file.txt	chmod 177 file.txt	--xrwxrwx
---------	chmod u+r,u-r,u=rw file.txt	chmod 600 file.txt	rw-------

4 = read 2 = write 1 = execute

04.06 chown/chgrp - changing ownership

Change who owns the file and the group:

```
chown owner:group file …
chown owner file …
chown :group file …
```



Change the group the file belongs to to `group`.

```
chgrp group file ...
```

Example:

```
touch share_this_file.txt
chown tux:friends share_this_file.txt
ls -l
-rw------- 1 tux friends   0 Sep 12 08:15 share_this_file.txt
```

When to use which permissions?

04.07 Hiding your files from everyone

(1) Protect your files from everyone else

⇒ u=rwx,g=,o= (700)

⇒ u=rw,g=,o= (600)

(2) Protect your files from everyone else and make sure you don't overwrite them or allow execution (no side effects)

⇒ u=r,g=,o= (400)

Tip: chmod also works with wilcards!

04.07 Commonly used permissions

(3) Only you can modify files, others may still read them

⇒ u=rw,g=r,o=r (644)

(4) Only you have write access,
others can get information about & read your files

⇒ u=rwx,g=rx,o=rx (755)

04.07 Commonly used permissions

(5) Only you have read/write access, others may still lookup information on your files but not read them

⇒ u=rwx,g=x,o=x (711)

04.08 Default guide to chmod for files

world executables files	`u=rwx,g=rwx,o=rx`	`775`
executables by group only	`u=rwx,g=rx,o=`	`750`
group modifiable files	`u=rw,g=rw,o=`	`660`
world readable files	`u=rw,g=r,o=r`	`644`
group readable files	`u=rw,g=r,o=`	`640`
private files	`u=rw,g=,o=`	`600`
private executables	`u=rwx,g=,o=`	`700`

DON'T USE 777 or 666.
These permissions pose security risks!

04.08

chmod for directories:

DON'T delete the execute bit on your folders.

Why? => you can not anymore access them using `cd` or `ls`!

⇒ If it happens and **you** own the file,
you can fix this by chmod 700 path/

USE 700 (private), 711(traversable) or 755(readable) on directories.

Note: 770 is o.k. for shared folders

04.08 chmod for directories

Example:

`sealion` wants to access `/home/tux` (700) and run `cat /home/tux/tux_profile.txt` (644)

```
sealion@server:~$ ls -l /home/tux
ls: cannot open directory '/home/tux':
Permission denied
```

```
sealion@server:~$ cat
/home/tux/tux_profile.txt
cat: /home/tux/tux_profile.txt: Permission
denied
```

Explanation:

/home/tux has permissions 700

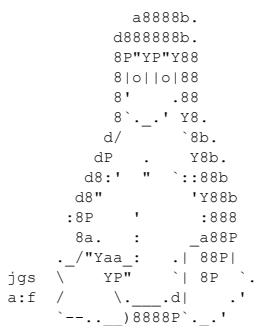
⇒ sealion has no read/execute permission, hence ls -l /home/tux produces Permission denied.
⇒ cat /home/tux/tux_profile.txt gives Permission denied too, because the location of tux_profile.txt can't be looked up because of the 700 permission on /home/tux

04.08 chmod for directories

Example:

`sealion` wants to access `/home/tux` (711) and run `cat /home/tux/tux_profile.txt` (644)

```
sealion@server:~$ ls -l /home/tux
ls: cannot open directory '/home/tux':
Permission denied

sealion@server:~$ cat
/home/tux/tux_profile.txt
Tux
```



Explanation:

/home/tux has permissions 711

⇒ sealion has no read so ls fails. However, sealion can cd into /home/tux!

⇒ cat /home/tux/tux_profile.txt works, because sealion can lookup file location for /home/tux.

⇒ 711 useful to allow content access of files but no traversal of directories!

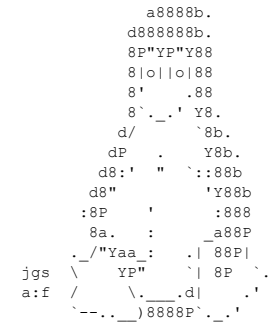
04.08 chmod for directories

Example:

`sealion` wants to access `/home/tux` (755) and run `cat /home/tux/tux_profile.txt` (644)

```
sealion@server:/home/tux$ ls -l /home/tux/
total 8
-rwxrwxrwx 1 tux tux 538 Sep 11 19:44 tux_profile.txt
-rwx------ 1 tux tux  96 Sep 11 18:41 tux_secret.txt
```

```
sealion@server:~$ cat /home/tux/tux_profile.txt
Tux
```



Explanation:

/home/tux has permissions 755

⇒ sealion read to both dir and file

⇒ 755 allows access & browsing.

What about 777 for directories?

Just Don't.

One more thing...

Special linux permissions

04.09 Special file permissions

Besides the permission for user/group/other, Linux has 3 special permissions which can be combined:

permission	octal	symbol	meaning
`setuid`	`4`	`s`	Allows a process to run as the owner of the file, not the user executing it
`setgid`	`2`	`s`	Allows a process to run with the group of the file, not the group of the user executing it
`sticky bit`	`1`	`t`	prevents a user from deleting another user's files even if they would normally have permission to do so

04.09 Special file permissions

Examples:

`chmod +t file.txt` ⇒ sets sticky bit for file.txt

`chmod g+s file.txt` ⇒ sets sgid bit for file.txt

`chmod u+s file.txt` ⇒ sets suid bit for file.txt

04.09 Special file permissions

`ls -l` for special permissions:

setuid	**setgid**	**sticky bit**
permission has S where execute bit x is normally located for **user**, s if execute bit x for user is also set for a file.	permission has S where execute bit x is normally located for **group**, s if execute bit x for group is also set for a file.	permission has T where execute bit x is normally located for **other**, t if execute bit x is also set for a file.

Examples:

```
chmod 1611 file.txt      ⇒ -rw---x--t   (sticky bit)
chmod 2644 file.txt   ⇒ -rw-r-Sr--   (setgid)
chmod 4400 file.txt   ⇒ -r-S------   (setuid)
chmod 7777 file.txt      ⇒ -rwsrwsrwt   (ALL permissions set)
```

04.09 Special file permissions

Why are they needed?

sticky bit:

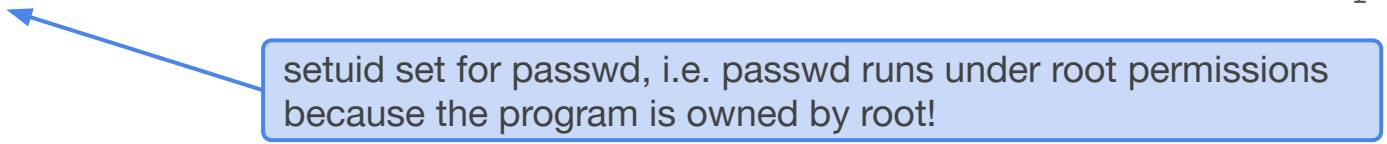
⇒ prevents other users from deleting files/directories in a public folder. E.g., /tmp where all users store temporary files.

04.09 Special file permissions

Why are they needed?

setuid:
⇒ passwd allows to change the password for a user. However, passwords need to be stored somewhere in a file. With setuid the program passwd runs with root privileges, but the user has no access to the password file.

```
ls -ls /usr/bin/passwd

-rwsr-xr-x 1 root root 59640 Mar 22 19:05 /usr/bin/passwd
```

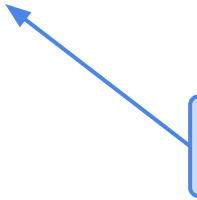


04.09 Special file permissions

Why are they needed?

setgid:
⇒ Files created in a shared folder which has the setgid bit set will belong to the group the folder belongs to.

```
ls /

drwxrws---    2 sealion friends  4096 Sep 12 02:05 recipes
```



/recipes is a shared folder between tux and sealion. Files created there will belong to friends!

links

04.10 Links

Links are special files which point to another file (in the wider sense).

`ln -s target link_name`

creates a symbolic link `link_name` pointing to target (Note the order!)

04.10 Links

Example:

Assuming we are in Tux's home directory (`pwd` ⇒ `/home/tux`), we could create a shortcut to work with Sealion's directory:

```
ln -s /home/sealion sl

ls -l

lrwxrwxrwx 1 tux tux      14 Sep 12 02:46 sl -> /home/sealion/

cd sl

-bash: cd: sl: Permission denied
```

permissions do not matter for the link. It's a pointer. When used, the target's permissions will be checked.

a link is a pointer, thus you see `link_name -> target` here!

l for link!

04.10 Links

Advice on links:
The link command is very powerful. If you have any doubt on how to use it, use per default `ln -s .`
Always check the order first!
Other options may break your system if you don't know what you're doing.

05 Streams & Pipes

CS6 Practical System Skills

Fall 2019

Leonhard Spiegelberg *lspiegel@cs.brown.edu*

Single commands are great...

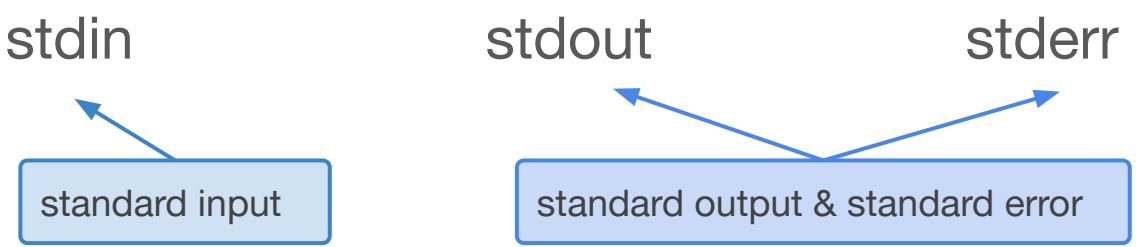
... but how about combining them?

05.01 Streams

Where do commands get their input?

Where do commands send their output?

⇒ two special files where output is sent to and one special file where input is read from:



05.01 Streams

⇒ A stream is a sequence of characters

⇒ Each of the three streams is identified by a unique file descriptor (number)

⇒ I.e. streams are actually a special type of file!

Stream	file descriptor
stdin	0
stdout	1
stderr	2

05.01 stdout & stderr

What is happening when we run a command?

⇒ `ls /home/sealion` with sufficient permissions will print its output to stdout which in turn is displayed by the terminal.

⇒ Without the permissions, an error message will be print to stderr (displayed by the terminal too).

05.02 Standard I/O redirection: output

⇒ Unix allows you to redirect streams from one file to another

`n> file`

`n` can be omitted, then it defaults to stdout.
I.e. `cmd > file` writes stdout of `cmd` to `file`!

redirects output from file descriptor `n` to a `file`, overwrites it if file exists.

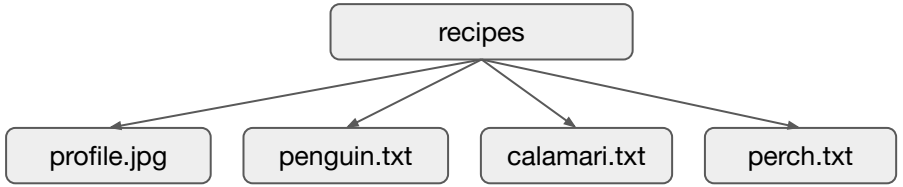
`n>> file`

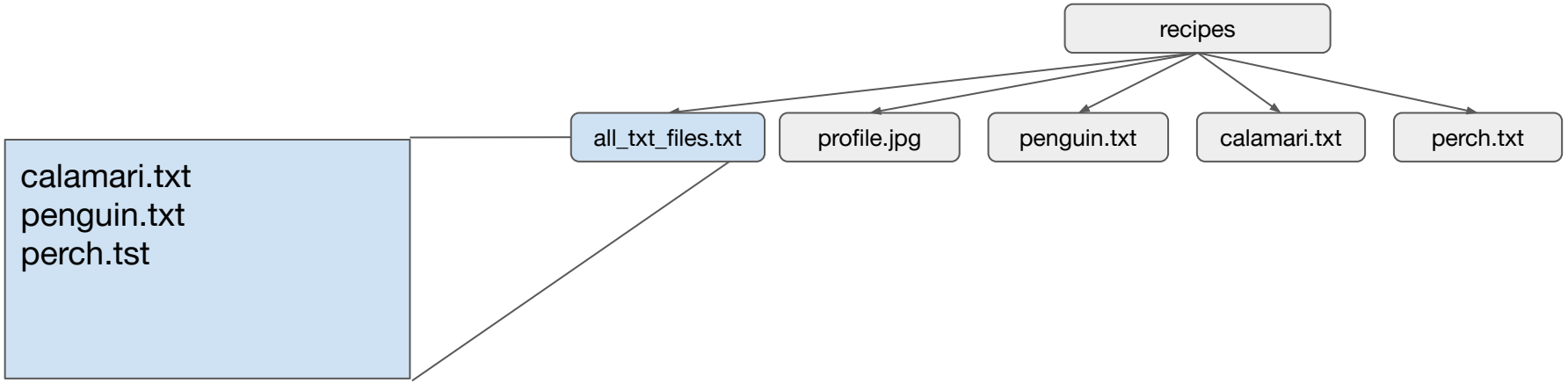
redirects output from file descriptor `n` to a `file`. If file doesn't exist, creates it, else content is appended.

05.02 Output redirection example

cwd is recipes

```
ls *.txt > all_txt_files.txt
```

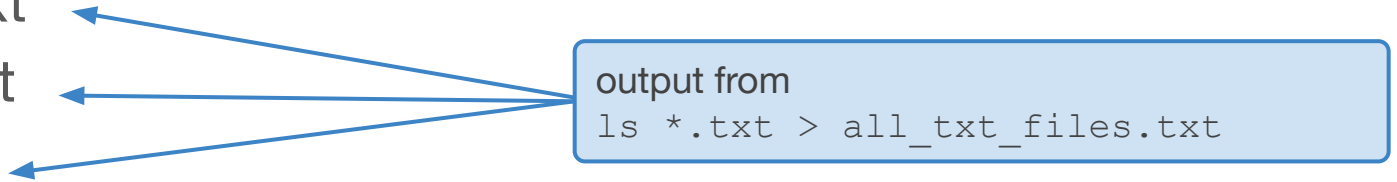




05.02 Output redirection example

Running `ls *.txt >> all_txt_files.txt` then,
will set the contents of all_txt_files.txt to:

calamari.txt
penguin.txt
perch.txt
all_txt_files.txt
calamari.txt
penguin.txt
perch.txt



05.02 Output redirection

More examples:

`ls ~ > /dev/null`	redirects stdout to special file /dev/null which discards data
`mkdir /data 2> mkdir_err_log.txt`	redirects stderr to mkdir_err_log.txt (run as regular user without privileges on /)

`cat > write_to_me.txt`
`1`
`2`
`3`
`Ctrl-d`
	redirects stdout to write_to_me.txt. cat without param allows to interactively write input, stop input mode by pressing Ctrl and d

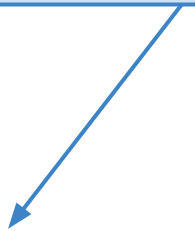
`cat /home/tux/tux_secret.txt 2> err.txt > stolen_secret.txt`	tries to access Tux's secret file tux_secret.txt (protected through file permissions!) ⇒ error gets written to err.txt, no output to stolen_secret.txt (empty file)

05.03 A new command - echo!

`echo [STRING]`

prints a new line, containing STRING if provided.

surround your text with ' ' or " ". More on these next lecture.



Examples:

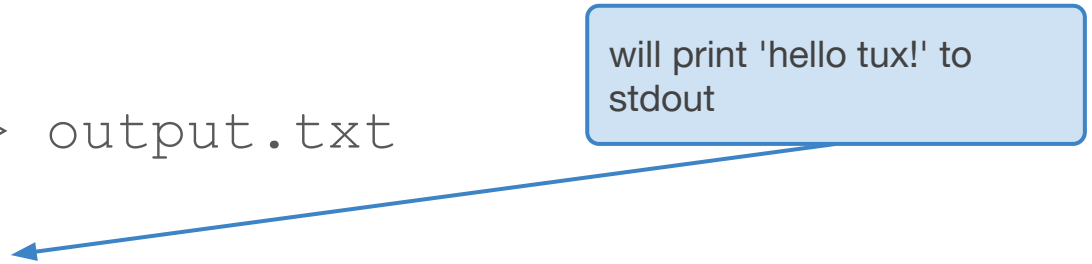
```
sealion@server:~$ echo Tux is a penguin
Tux is a penguin
sealion@server:~$ echo "usually afraid of seals and sealions"
usually afraid of seals and sealions
sealion@server:~$ echo 'but became friends with sealion!'
but became friends with sealion!
sealion@server:~$ echo "isn't that great?"
isn't that great?
```

05.04 Input redirection

less commonly used than output redirection.

`man cat` ⇒ when cat has no argument, it reads its input from stdin

```
echo 'hello tux!' > output.txt

cat < output.txt
```



05.05 Some basic text processing commands

`sort [file]`	sorts lines of `file`, or stdin if no input is given
`head [file]`	prints per default first 10 lines of `file` to stdout, or stdin if no input is given. Use -n <count> to print <count> lines, -c <count> to print <count> bytes
`tail [file]`	same as head, just takes the last lines (also with -n / -c)

05.06 Building pipelines

prints like in C / Java / Python a formatted string to stdout. I.e. stdout will have sealion, tux, penguin, crabby each on one line. (\n is the newline character

```
printf "sealion\ntux\npenguin\ncrabby" > temp.txt

sort temp.txt > temp2.txt

head -n 3 temp2.txt > result.txt

rm temp.txt

rm temp2.txt
```

Can we do better?

05.06 Building pipelines

rewriting commands to use stdin and feeding them temp files

```
printf "sealion\ntux\npenguin\ncrabby" > temp.txt

sort < temp.txt > temp2.txt

head -n 3 < temp2.txt > result.txt

rm temp.txt

rm temp2.txt
```

Can we do even better?

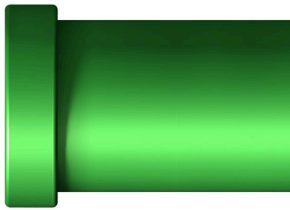
05.07 Pipes

```
cmd1 | cmd2
```

pipe operator `|` ⇒ connects stdout of `cmd1` to stdin of `cmd2`

⇒ allows you to get rid of temporary files

Example:



```
printf "sealion\ntux\npenguin\ncrabby" |
sort |
head -n 3 > result.txt
```

End of lecture.

Next class: Tue, 4pm-5:20pm @ CIT 477