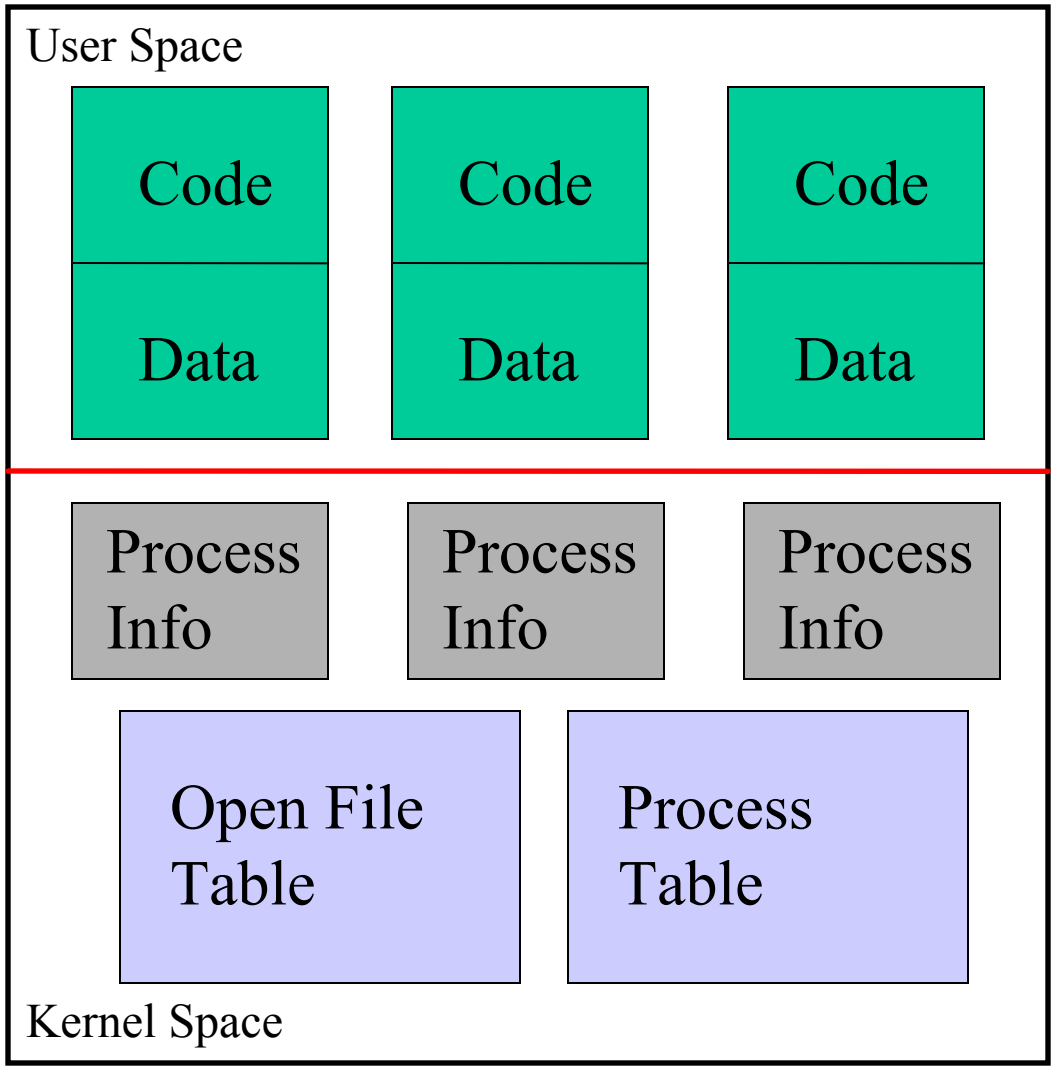
Lecture 3

Processes and Filters

Kernel Data Structures

- Information about each process.
- **Process table**: contains an entry for every process in the system.
- **Open-file table**: contains at least one entry for every open file in the system.



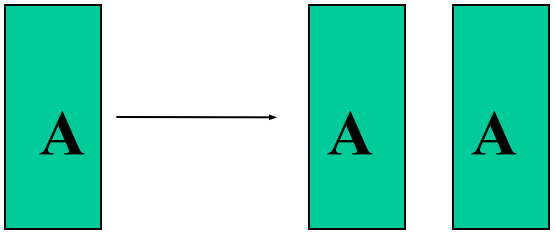
Unix Processes

Process: An entity of execution

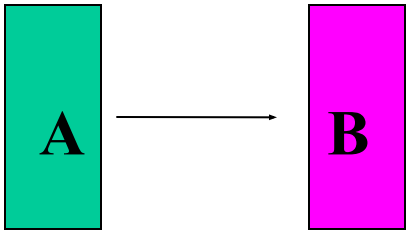
- *Definitions*
  - **program**: collection of bytes stored in a file that can be run
  - **image**: computer execution environment of program
  - **process**: execution of an image
- Unix can execute many processes simultaneously.

Process Creation

- Interesting trait of UNIX
- **fork** system call clones the current process



- **exec** system call reinitializes current process with new image



- A **fork** is typically followed by an **exec**

Process Setup

- All of the per process information is copied with the **fork** operation
  - Working directory
  - Open files
- *Copy-on-write* makes this efficient
- Before **exec**, these values can be modified
- Both processes receive return from **fork()**
  - Child gets 0; parent gets PID of child

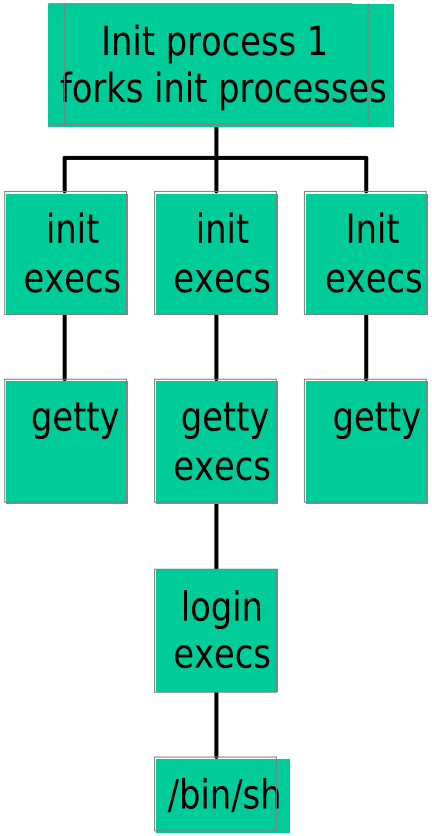
fork and exec

Example: the shell

```
while(1) {
  display_prompt();
  read_input(cmd, params);
  pid = fork();        /* create child */
  if (pid != 0)
    waitpid(-1, &stat, 0);  /* parent waits */
  else
    execve(cmd, params, 0); /* child execs */
}
```

Unix process genealogy

Process generation



Background Jobs

- By default, executing a command in the shell will wait for it to exit before printing out the next prompt
- Trailing a command with & allows the shell and command to run simultaneously

```
$ /bin/sleep 10 &
[1] 3424
$
```

Program Arguments

- When a process is started, it is sent a list of strings
  - **`argv`**, **`argc`**
- The process can use this list however it wants to

`find . -name 'foo*'`

0	`find`
1	`.`
2	`-name`
3	`foo*`

Ending a process

- When a process ends, there is a return code associated with the process
- This is a positive integer:
  - 0 means success
  - > 0 represents various kinds of failure.
    Up to process to decide meaning.

Process Information Maintained

- Working directory
- File descriptor table
- Process id
  - number used to identify process
- Process group id
  - number used to identify set of processes
- Parent process id
  - process id of the process that created the process

Process Information Maintained

- Umask
  - Default file permissions for new file

We haven't talked about these yet:

- Effective user and group id
  - The user and group this process is running with permissions as
- Real user and group id
  - The user and group that invoked the process
- Environment variables

Setuid and Setgid Mechanisms

- The kernel can set the effective user and group ids of a process to something different than the real user and group
  - Files executed with a setuid or setgid flag set cause the these values to change
- Make it possible to do privileged tasks:
  - Change your password
- Open up a can of worms for security if buggy

Environment of a Process

- A set of name-value pairs associated with a process
- Keys and values are strings
- Passed to children processes
- Cannot be passed back up
- Common examples:
  - **`PATH`**: Where to search for programs
  - **`TERM`**: Terminal type

The PATH environment variable

- Colon-separated list of directories.
- Non-absolute pathnames of executables are only executed if found in the list.
  - Searched left to right

- Example:

```
$ myprogram
sh: myprogram not found
$ PATH=/bin:/usr/bin:/home/kornj/bin
$ myprogram
hello!
```

Shell Variables

- Shells have several mechanisms for creating variables. A variable is a name representing a string value. Example: **PATH**
  - Shell variables can save time and reduce typing errors
- Allows you to store and manipulate information
  - Eg: `ls $DIR > $FILE`
- Two types: **local** and **environment**
  - *local* are set by the user or by the shell itself
  - *environment* come from the operating system and are passed to children

Variables (con't)

- Syntax varies by shell

```
varname=value          # sh, ksh, bash
set varname = value    # csh
```

- To access the value: **`$varname`**

- Turn local variable into environment:

```
export varname         # sh, ksh, bash
setenv varname value # csh
```

Environmental Variables

NAME	MEANING
`$HOME`	Absolute pathname of your home directory
`$PATH`	A list of directories to search for
`$MAIL`	Absolute pathname to mailbox
`$USER`	Your user id
`$SHELL`	Absolute pathname of login shell
`$TERM`	Type of your terminal
`$PS1`	Prompt

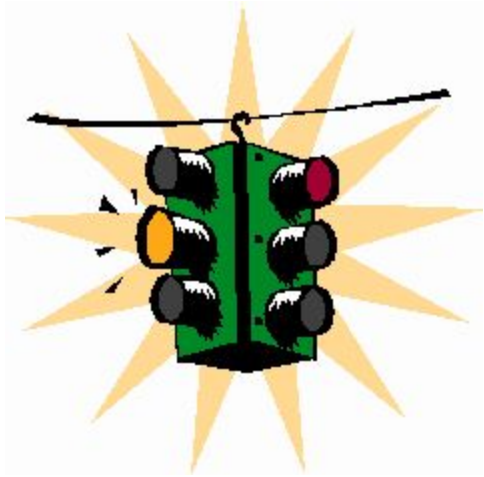
Inter-process Communication

Ways in which processes communicate:

- Passing arguments, environment
- Read/write regular files
- Exit values
- **Signals**
- **Pipes**

Signals

- **Signal**: A message a process can send to a process or process group, if it has appropriate permissions.
- Message type represented by a symbolic name
- For each signal, the **receiving process** can**:**
  - Explicitly ignore signal
  - Specify action to be taken upon receipt (**signal handler**)
  - Otherwise, default action takes place (usually process is killed)
- Common signals:
  - SIGKILL, SIGTERM, SIGINT
  - SIGSTOP, SIGCONT
  - SIGSEGV, SIGBUS



An Example of Signals

- When a child exits, it sends a **SIGCHLD** signal to its parent.
- If a parent wants to be notified that a child has exited, it tells the system it wants to catch the **SIGCHLD** signal, via **waitpid()**
- Default action for **SIGCHLD** signal is to ignore it

Process Subsystem utilities

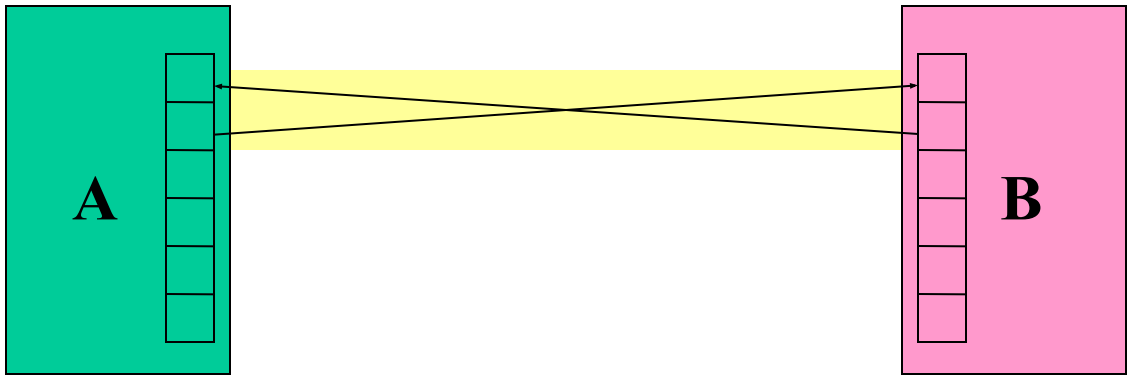
ps	monitors status of processes
kill	send a signal to a pid
wait	parent process wait for one of its children to terminate
nohup	makes a command immune to the hangup and terminate signal
sleep	sleep in seconds
nice	run processes at low priority

Pipes

One of the cornerstones of UNIX

Pipes

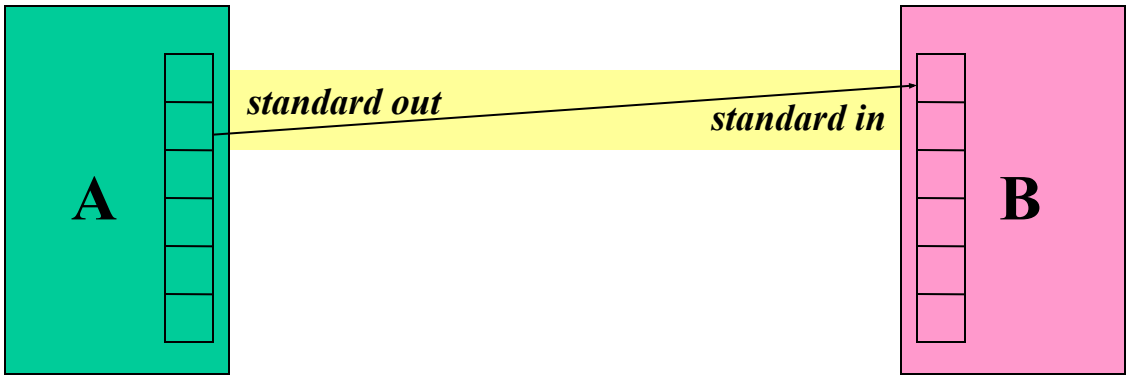
- General idea: The input of one program is the output of the other, and vice versa



- Both programs run at the same time

Pipes (2)

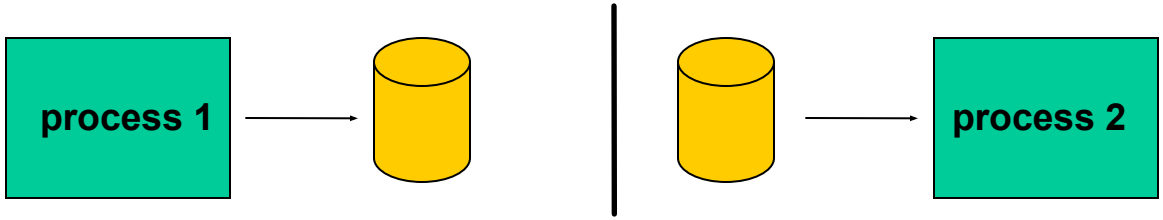
- Often, only one end of the pipe is used



- Could this be done with files?

File Approach

- Run first program, save output into file
- Run second program, using file as input



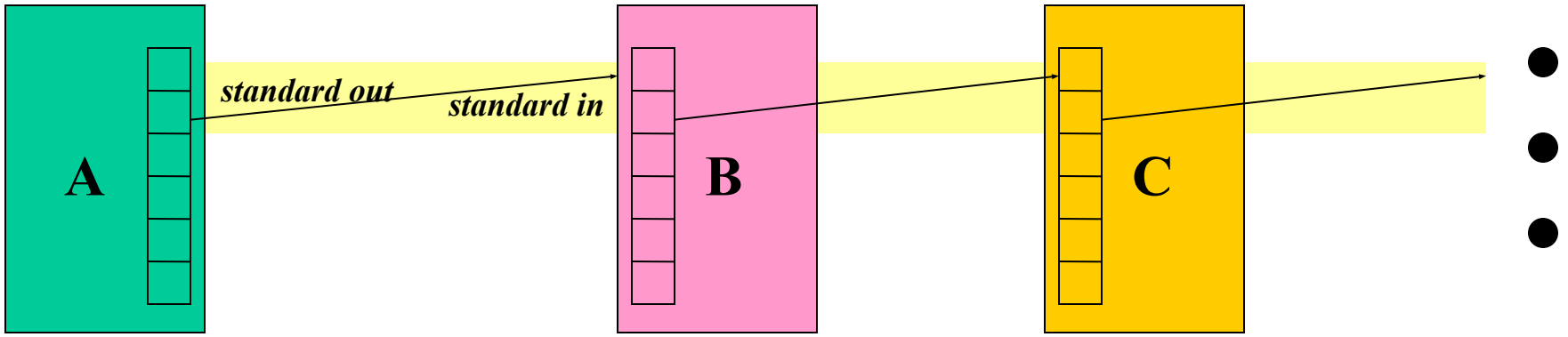
- Unnecessary use of the disk
  - Slower
  - Can take up a lot of space
- Makes no use of multi-tasking

More about pipes

- When process tries to read from pipe but nothing is available…
  - UNIX puts the reader to sleep until data available
- When reader cannot keep up with writer…
  - Unread data is buffered up to the *pipe size*
  - If pipe fills up,writer put to sleep until the reader frees up space (by reading)
- Multiple readers and writers possible with pipes.

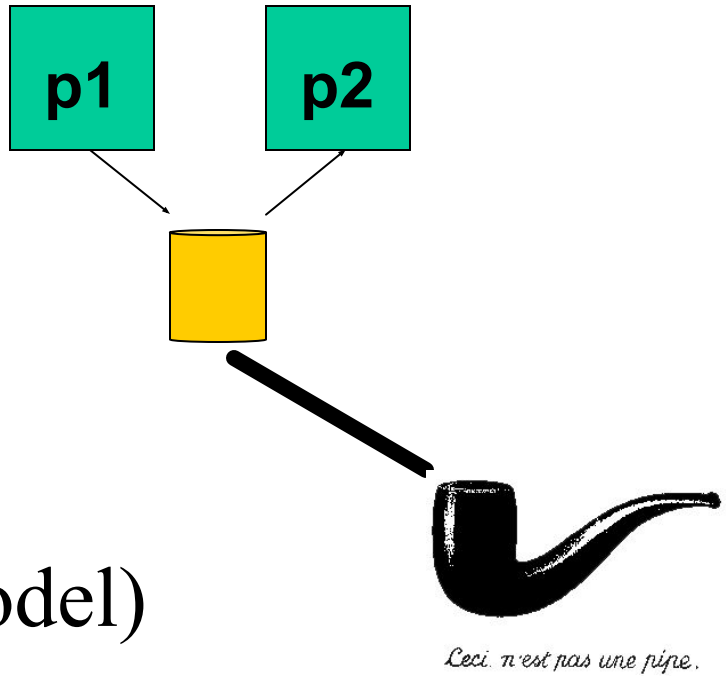
More about Pipes

- Pipes are often chained together
  - Called *filters*



Interprocess Communication For Unrelated Processes

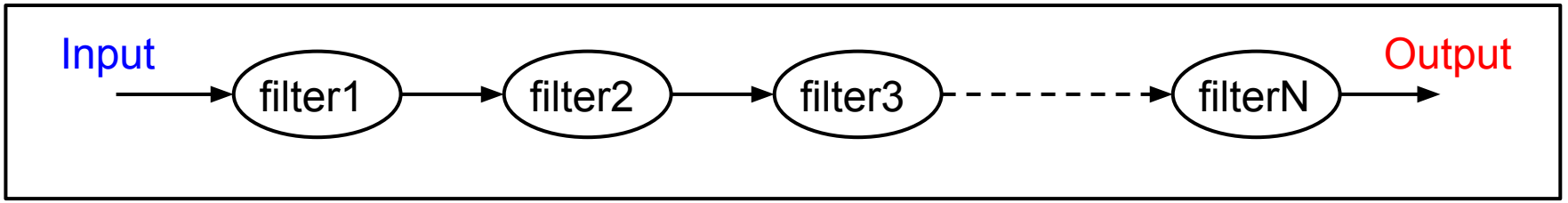
- FIFO (*named pipes*)
  - A special file that when opened represents pipe
- System V IPC
  - message queues
  - semaphores
  - shared memory
- Sockets (client/server model)



Pipelines

- Output of one program becomes input to another
  - Uses concept of UNIX **pipes**
- Example: **`$ who | wc -l`**
  - counts the number of users logged in
- Pipelines can be long

`filter1 | filter2 | filter3 | ... | filterN`



What's the difference?

Both of these commands send input to ***command*** from a file instead of the terminal:

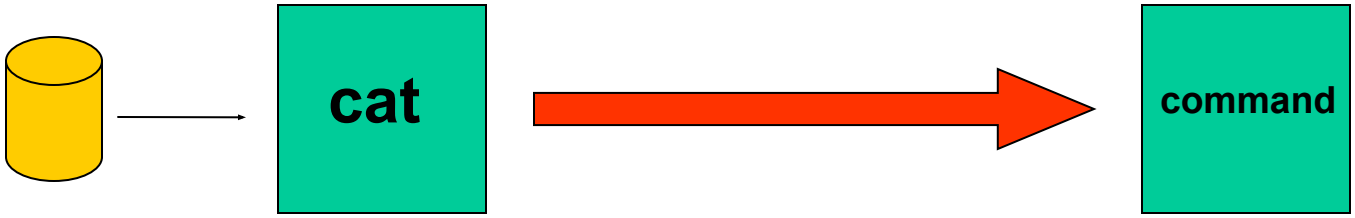
```
$ cat file | command
```

vs.

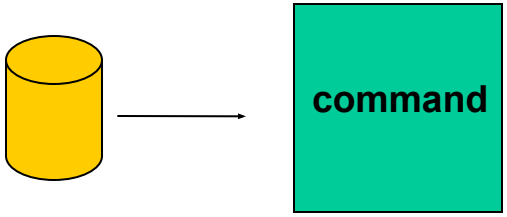
```
$ command < file
```

An Extra Process

```
$ cat file | command
```

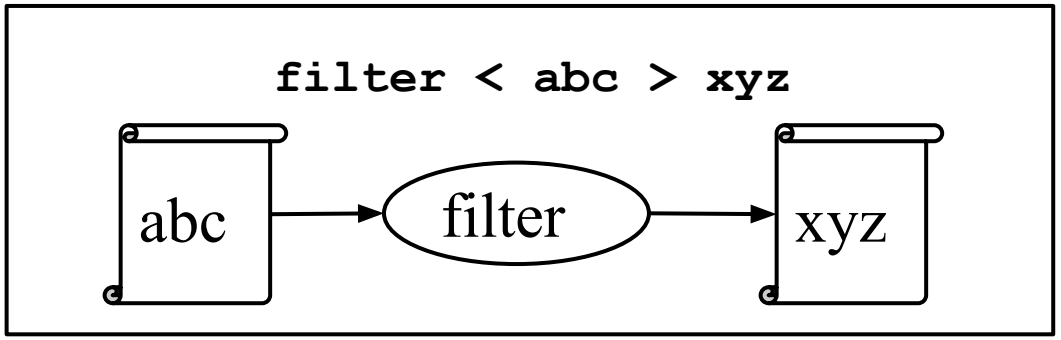


```
$ command < file
```



Introduction to Filters

- A class of Unix tools called ***filters***.
  - Utilities that read from standard input, transform the file, and write to standard out
- Using filters can be thought of as *data oriented programming*.
  - Each step of the computation transforms data *stream*.



Examples of Filters

- **sort**
  - Input: lines from file/stdin
  - Output: lines sorted
- **grep**
  - Input: lines from file/stdin
  - Output: lines that match pattern as argument
- **awk**
  - Programmable filter

cat: The simplest filter

- The **`cat`** command copies its input to output unchanged (*identity filter*). When supplied a list of file names, it con**cat**enates them onto stdout.
- Some options:
  - **`-n`** **n**umber output lines (starting from 1)
  - **`-v`** display control-characters in **v**isible form (e.g. ^C)

```
cat file*
ls | cat -n
```

head

- Display the first few lines of a specified file
- Syntax: *head [-n] [filename...]*

  -n - number of lines to display, default is 10

  filename... - list of filenames to display
- When more than one filename is specified, the start of each files listing displays

```
==> filename <==
```

tail

- Displays the last part of a file
- Syntax: *tail +|-number [lbc] [f] [filename]*
  or: *tail +|-number [l] [rf] [filename]*

  +number - begins copying at distance *number* from beginning of file, if *number* isn't given, defaults to 10

  -number - begins from end of file

  l,b,c - *number* is in units of lines/block/characters

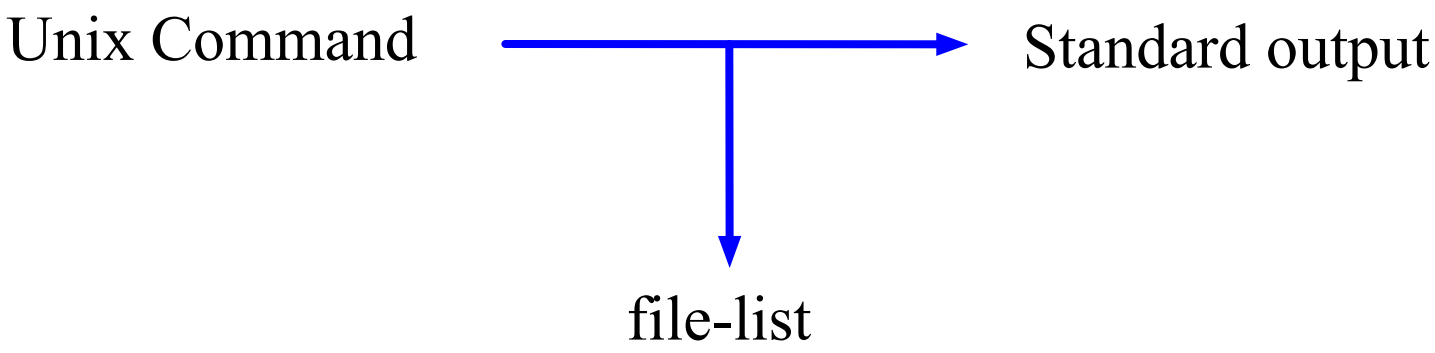
  r - print in reverse order (lines only)

  f - if input is not a pipe, do not terminate after end of file has been copied but loop. This is useful to monitor a file being written by another process

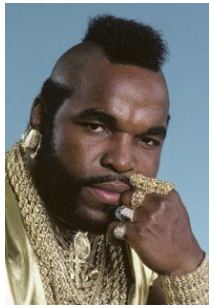
head and tail examples

```
head /etc/passwd

head *.c

tail +20 /etc/passwd

ls -lt | tail -3

head -100 /etc/passwd | tail -5

tail -f /var/httpd/access_log
```

tee



- Copy standard input to standard output and one or more files
  - Captures intermediate results from a filter in the pipeline



tee (2)

- Syntax: *tee [-ai] file-list*

  -a - append to output file rather than overwrite, default is to overwrite (replace) the output file

  -i - ignore interrupts

  file-list - one or more file names for capturing output

- Examples:

```
ls | head -10 | tee first_10 | tail -5
who | tee user_list | wc -l
```

Unix Text Files: Delimited Data

Tab Separated

```
John    99
Anne    75
Andrew 50
Tim 95
Arun    33
Sowmya 76
```

Pipe-separated

```
COMP1011|2252424|Abbot, Andrew John |3727|1|M
COMP2011|2211222|Abdurjh, Saeed |3640|2|M
COMP1011|2250631|Accent, Aac-Ek-Murhg |3640|1|M
COMP1021|2250127|Addison, Blair |3971|1|F
COMP4012|2190705|Allen, David Peter |3645|4|M
COMP4910|2190705|Allen, David Pater |3645|4|M
```

Colon-separated

```
root:ZHolHAHZw8As2:0:0:root:/root:/bin/ksh
jas:nJz3ru5a/44Ko:100:100:John Shepherd:/home/jas:/bin/ksh
cs1021:iZ3sO90O5eZY6:101:101:COMP1021:/home/cs1021:/bin/bash
cs2041:rX9KwSSPqkLyA:102:102:COMP2041:/home/cs2041:/bin/csh
cs3311:mLRiCIvmtI9O2:103:103:COMP3311:/home/cs3311:/bin/sh
```

cut: select columns

- The **`cut`** command prints selected parts of input lines.
  - can select columns (assumes tab-separated input)
  - can select a range of character positions
- Some options:

  -f *listOfCols***:** print only the specified columns (tab-separated) on output

  -c *listOfPos***:** print only chars in the specified positions

  -d *c***:** use character *c* as the column separator
- Lists are specified as ranges (e.g. `1-5`) or comma-separated (e.g. `2,4,5`).

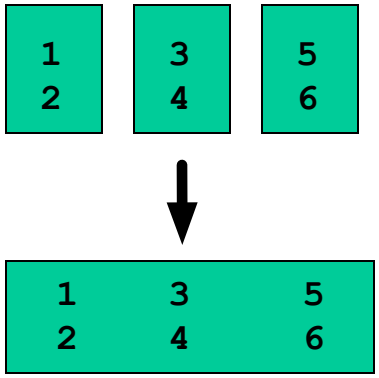
cut examples

```
cut -f 1 < data

cut -f 1-3 < data

cut -f 1,4 < data

cut -f 4- < data

cut -d'|' -f 1-3 < data

cut -c 1-4 < data
```

Unfortunately, there's no way to refer to "last column" without counting the columns.

paste: join columns

- The **`paste`** command displays several text files "in parallel" on output.
- If the inputs are files **a**, **b**, **c**
  - the first line of output is composed of the first lines of **a**, **b**, **c**
  - the second line of output is composed of the second lines of **a**, **b**, **c**



- Lines from each file are separated by a tab character.
- If files are different lengths, output has all lines from longest file, with empty strings for missing

paste example

```
cut -f 1 < data > data1

cut -f 2 < data > data2

cut -f 3 < data > data3

paste data1 data3 data2 > newdata
```

sort: Sort lines of a file

- The **`sort`** command copies input to output but ensures that the output is arranged in ascending order of lines.
  - By default, sorting is based on ASCII comparisons of the whole line.
- Other features of **`sort`**:
  - understands text data that occurs in columns.
    (can also sort on a column other than the first)
  - can distinguish numbers and sort appropriately
  - can sort files *in place* as well as behaving like a filter

sort: Options

- Syntax: *sort [-dftnr] [-o filename] [filename(s)]*
  - *-d* Dictionary order, only letters, digits, and whitespace are significant in determining sort order
  - *-f* Ignore case (fold into lower case)
  - *-t* Specify delimiter
  - *-n* Numeric order, sort by arithmetic value instead of first digit
  - *-r* Sort in reverse order
  - *-o* *filename* - write output to filename, filename can be the same as one of the input files
- Lots of more options…

sort: Specifying fields

- Delimiter : **`-t`***d*
- Ancient way:
  - **`+`***f*`[.`*c*`][`*options*`]  [-`*f*`[.`*c*`][`*options*`]`
    `+1.1 -2 +0 -2 +3n`
  - Exclusive
  - Start from 0 (unlike cut, which starts at 1)
- Modern way:
  - **`-k`** *f*`[.`*c*`][`*options*`][,`*f*`[.`*c*`][`*options*`]]`
    `-k2.2,2 -k1,2 -k4n`
  - Inclusive
  - Start from 1

sort Examples

```
sort +2nr < data
sort -k2nr data

sort -t: -k5 /etc/passwd

sort -o mydata mydata
```

uniq: list unique items

- Remove or report adjacent duplicate lines
- Syntax: *uniq [-cdu] [input-file] [output-file]*
  - **`-c`** Supersede the -u and -d options and generate an output report with each line preceded by an occurrence count
  - **`-d`** Write only the duplicated lines
  - **`-u`** Write only those lines which are not duplicated
- The default output is the union (combination) of *-d* and *-u*

wc: Counting results

- The word count utility, **wc**, counts the number of lines, characters or words
- Options:
  - `-l` Count lines
  - `-w` Count words
  - `-c` Count characters
- Default: count lines, words and chars

wc and uniq Examples

```
who | sort | uniq -d

wc my_essay

who | wc

sort file | uniq | wc -l

sort file | uniq -d | wc -l

sort file | uniq -u | wc -l
```

tr: translate characters

- Copies standard input to standard output with substitution or deletion of selected characters
- Syntax: *tr [-cds] [string1] [string2]*
  - `-d` delete all input characters contained in *string1*
  - `-c` complements the characters in *string1* with respect to the entire ASCII character set
  - `-s` squeeze all strings of repeated output characters in the last operand to single characters

tr (continued)

- *tr* reads from standard input (only).
  - Any character that does not match a character in *string1* is passed to *standard output* unchanged
  - Any character that does match a character in *string1* is translated into the corresponding character in *string2* and then passed to *standard output*
- Examples

`tr s z`	replaces all instances of *s* with z
`tr so zx`	replaces all instances of *s* with *z* and *o* with *x*
`tr a-z A-Z`	replaces all lower case characters with upper case characters
`tr –d a-c`	deletes all a-c characters

tr uses

- Change delimiter

```
tr '|' ':'
```

- Rewrite numbers

```
tr ,. .,
```

- Import DOS files

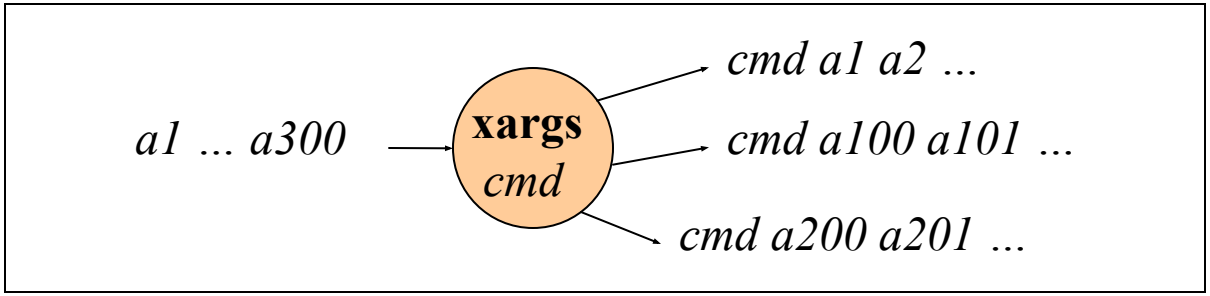
```
tr -d '\r' < dos_file
```

- Find printable ASCII in a binary file

```
tr -cd '\na-zA-Z0-9 ' < binary_file
```

xargs

- Unix limits the size of arguments and environment that can be passed down to child
- What happens when we have a list of 10,000 files to send to a command?
- **xargs** solves this problem
  - Reads arguments as standard input
  - Sends them to commands that take file lists
  - May invoke program several times depending on size of arguments



find utility and xargs

- **`find . -type f -print | xargs wc -l`**
  - **`-type f`** for files
  - **`-print`** to print them out
  - **xargs** invokes **wc** 1 or more times

```
wc -l a b c d e f g
wc -l h i j k l m n o
...
```

- Compare to:

```
find . -type f –exec wc -l {} \;
```

Next Time

- Regular Expressions
  - Allow you to search for text in files
  - **grep** command
- We will soon learn how to write *scripts* that use these utilities in interesting ways.