SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 3

Fabian Wermelinger
Harvard University
CS107 / AC207

Thursday, September 8th 2022

LAST TIME

- More on Linux commands and the `man`-pages
- Unix philosophy and pipes
- Regular expressions and `grep`
- File attributes and the `find` command
- Text editors and IDEs

TODAY

Main topics: ***Command line customization, I/O redirection, Environment variables and shell scripting, Process management***

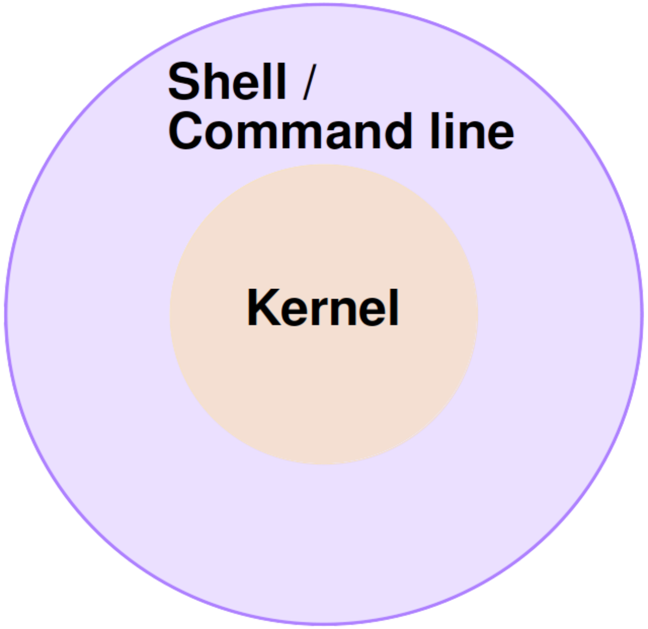
Details:

- Configuration files read by the shell at startup.
- Login shells and non-login shells.
- Redirection of file content and `stdout`/`stderr`.
- How to use environment variables and how to set them.
- Basics of writing shell scripts (same as typing commands in the shell but written in a file instead).
- Managing Jobs and processes in Linux, suspending and continuing execution.

LAB SECTIONS AND OFFICE HOURS

- You have been assigned to lab sections based on your preferences. You should have received an email notification if you submitted your preferences.
- Please check https://my.harvard.edu/ for your assigned lab section.
- If you were not assigned a lab section, please contact the teaching staff at cs107-staff@g.harvard.edu as soon as possible.
- You can find spreadsheets for time and location overview of office hours and lab sections in the class repository:
    - https://code.harvard.edu/CS107/main/blob/master/office_hours.xls
    - https://code.harvard.edu/CS107/main/blob/master/lab_groups.xls

RECALL THE SHELL IS YOUR COMMAND INTERFACE



- The tools of a carpenter are *essential* for his/her craft. They must be *sharp* for best results.
- Similar, the shell and editor are your tools. *Make them your own.*
- There are different shell interpreters:
  - `sh` (Bourne shell)
  - `bash` (Bourne Again Shell, Mac: since OSX Jaguar)
  - `csh` (C-shell)
  - `ksh` (Korn shell)
  - `zsh` (Mac: since OSX Catalina)
- Each of those shells executes a number of files at startup. You can use these files to run commands (`rc`) and configure your shell.

SHELL CUSTOMIZATION

- ***Examples for configuration:*** user prompt, environment variables, auto-completion, command aliases, color theme and appearance, message of the day (motd), ...
- The configuration is implemented in startup files that are read by the `bash` shell whenever it starts. *There are two types of shells which read the following files in the given order:*
  - Interactive login shell or with `--login` option:
    1. `/etc/profile`
    2. `~/.bash_profile` (if it exists, read and execute then stop)
    3. `~/.bash_login` (if it exists, read and execute then stop)
    4. `~/.profile` (if it exists, read and execute then stop)
  - Interactive non-login shell (e.g. a terminal emulator like `xterm`):
    1. `~/.bashrc`

SHELL CUSTOMIZATION

- Which files being read at shell startup *depend on the shell you are using*. See https://en.wikipedia.org/wiki/Unix_shell#Configuration_files for a good overview.
- Think of a ***login shell*** this way:
  - It is the *first* shell started when you login to the system. It must exist.
  - If you use Ubuntu with a GUI, you will not notice the login shell as the system boots directly into graphical mode.
  - On a headless server you will be dropped into a login shell. This is called an ***interactive login shell***.
  - From the login shell you can create other shell instances, these are ***interactive non-login*** shells.

SHELL CUSTOMIZATION

***Summary for bash*:**

- Files read for interactive login shell:
  1. `/etc/profile`
  2. One of (in that order): `~/.bash_profile`, `~/.bash_login`, `~/.profile`
- Files read for interactive non-login shell:
  1. `~/.bashrc`

Typically, `~/.bash_profile` contains this code:

```
[[ -f ~/.bashrc ]] && source ~/.bashrc  # if ~/.bashrc exists, source its contents
```

Conclusion: to customize your `bash` shell, edit `~/.bashrc`. To customize your `zsh` shell, edit `~/.zshrc` instead.

Further information for startup files:
https://www.gnu.org/software/bash/manual/html_node/Bash-Startup-Files.html

SHELL CUSTOMIZATION

*A few comments about **bash** and **zsh**:*

- Mac users will most likely be working with `zsh`, a newer shell with some additional features.
- The default shell on Linux is `bash`. You must install `zsh` from the package repo if you want to use it on Linux.
- While startup files may be different, most *scripts* should run with either shell.
- You will be confronted with `bash` on most remote machines and servers. Keep that in mind when you work with `zsh` and must be compatible with `bash`.

Additional reading for Mac users:

- Moving to `zsh` - Scripting OSX
- What should/shouldn't go in `.zshenv`, `.zshrc`, `.zlogin`, `.zprofile`, `.zlogout`?
- About `.bash_profile` and `.bashrc` on MacOS

SHELL CUSTOMIZATION

Simple .bashrc example:

```
set -o vi # configure the shell prompt to behave like vi (normal and insert mode)
          # default is set -o emacs

alias diff='diff --color=always' # alias for the diff command to enforce color

# These are some environment variables.  The export keyword is important
export EDITOR=vim
export GIT_EDITOR=vim
export VISUAL=vim

# update the PATH environment variable
export PATH=$HOME/bin:$HOME/.local/bin:$HOME/go/bin:$PATH

# set a custom primary prompt: promt is defined in the variable PS1, see `man bash`
export PS1='\e[36m\w\e[0m\$ '
```

- A useful adaptive prompt for bash and zsh: https://github.com/nojhan/liquidprompt
- Shell color themes based on 16 colors: https://github.com/chriskempson/base16-shell
- There are many more online to be found via search engines.

EXAMPLES FOR SHELL CUSTOMIZATION

- You find yourself often typing `ls -l`. Create an *alias* named `ll` to minimize your future typing overhead. (See previous slide for an example `alias`.)
- Figure out what the prompt from the previous slide is doing:

```
# set a custom primary prompt: promt is defined in the variable PS1, see `man bash`
export PS1='\e[36m\w\e[0m\$ '
```

  Configure your own prompt. This page might be helpful.
- Change the interpretation mode to `vi` with

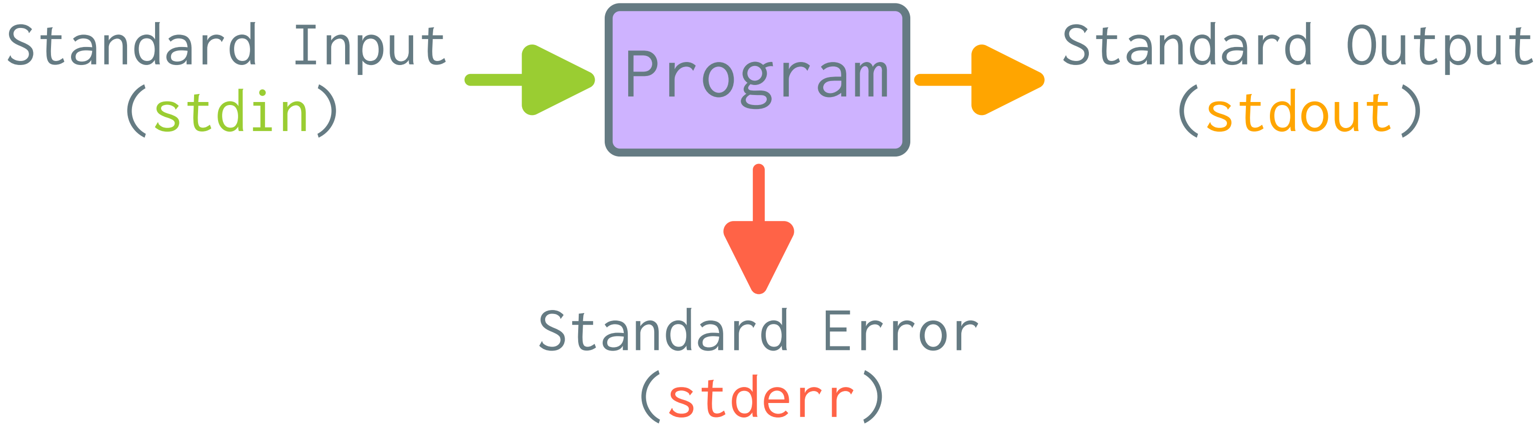
```
$ set -o vi
```

  Your shell now operates with *normal* and *insert* modes, similar to `vi` or `vim`. You can change back to non-modal interpretation with

```
$ set -o emacs
```

  which is the default.

INPUT/OUTPUT (I/O) AND REDIRECTION



- Inside the shell, `stdin` is received from your keyboard
- A program generates *two output streams*:
  1. `stdout`: normal program output
  2. `stderr`: output associated to something gone wrong *(e.g. compiler warning or error)*
- Recall the file descriptors: `stdin=0`, `stdout=1`, `stderr=2`
- To send and EOF (end-of-file) character, press `ctrl-d`

REDIRECTION OF I/O STREAMS

- You have learned about the Unix pipe "`|`" which redirects the `stdout` of a program into the `stdin` for the next program in the pipeline (see the "`SHELL GRAMMAR`" section in `man bash`)
- You can also *redirect* any data stream ***to or from files***.
- To redirect `stdout` to a file use the "`>`" operator:

```
$ ls -l > ls_long_output  # redirect output into file
```

- To redirect file contents to `stdin` use the "`<`" operator:

```
$ sort < some_data  # input content of some_data into sort
```

- You can combine both:

```
$ sort < some_data > some_sorted_data
```

Or equivalently using a pipe (the above is more compact)

```
$ cat some_data | sort > some_sorted_data
```

REDIRECTION OF I/O STREAMS

- You can either ***create (or overwrite)*** files or ***append*** to existing files:
  - Use `>` to ***create or overwrite*** (it will delete previous contents)
  - Use `>>` to append to existing files (ideal for logging)
- There are *two special data sinks* in Linux:
  1. `/dev/null`: data written to this device is *discarded*. Reading from this device always returns the end-of-file (EOF) character.
  2. `/dev/zero`: data written to this device is *discarded*. Reading from this device returns the `'\0'` (NUL) byte (see ASCII table).

Example: filter spam email and send it to `/dev/null`

```
$ script_to_filter_spam_email >/dev/null   # /dev/null behaves like a black hole
```

REDIRECTION OF I/O STREAMS

- File redirection operates on `stdout` by default.
- You can specify the *file descriptor* explicitly.
  Example: to redirect `stderr` use file descriptor 2:

```
$ my_prog 2> error_log  # only redirect stderr
```

- For convenience, you can redirect both `stdout` and `stderr` at the same time using the "&" descriptor:

```
$ my_prog &> full_log  # redirect stdout and stderr
```

- This also works if you want to use pipes (*by default only `stdout` is piped into `stdin`*):

```
$ prog1 |& prog2  # pipe stdout and stderr into stdin of prog2
```

- You can also chain redirection operators:

```
$ my_prog > my_output 2> my_error_log  # write error log file
$ my_prog > my_output 2>&1  # redirect stderr to stdout instead of separate file
```

ASIDE: WHITE SPACE IN FILENAMES

- We have seen the wildcard "*" that matches every character (even white space). ***For example:*** list all python files

```
$ ls
exercise_1.py  exercise_2.py  README.md
$ ls *.py
exercise_1.py  exercise_2.py
```

- ***White space in filenames:*** although common on Windows, it is *bad practice* to create filenames with spaces. In the shell, white space separates arguments to commands and you must take special care when you parse filenames. ***Example:***

```
$ touch exercise 1.py; ls  # touch creates 2 files: 1.py and exercise
1.py  exercise  README.md
$ touch exercise\ 1.py; ls  # you must escape white space to get a single file
1.py   exercise  'exercise 1.py'   README.md
$ touch 'exercise 2.py'; ls  # or pass a string
1.py   exercise  'exercise 1.py'  'exercise 2.py'  README.md
```

ENVIRONMENT VARIABLES

- You can customize your environment by setting the values of certain *environment variables*.
- You have already seen them when customizing your prompt by setting the value of `PS1`, which is an environment variable.
- You can get a list of all environment variables and their corresponding value with the `env` command
- Any shell variables (not only environment variables) can be *dereferenced* by prefixing them with a "`$`" character:

```
$ my_var='Hello CS107/AC207!'
$ echo $my_var
Hello CS107/AC207!
$ echo my_var
my_var
$ echo $HOME  # Environment variables are usually written in ALL CAPS
/home/fabs
```

THE PATH ENVIRONMENT VARIABLE

The role of the PATH environment variable is to specify the search path(s) used by the shell to find executable programs.

- For every command you enter, the shell checks if this command is a *built-in* command (see the "`SHELL BUILTIN COMMANDS`" in `man bash`).
- If it is not built-in, the shell will check the path(s) defined in the PATH environment variable to see whether it can find the executable in these locations.
- Finally, the shell will give up:

```
$ this_command_does_not_exist
bash: this_command_does_not_exist: command not found
```

- If a command is not reachable via `PATH` but the *path to the executable is specified instead*, the shell will execute the command (see lecture codes):

```
$ ./this_command_exists_locally  # note the leading path `./`
You have executed ./this_command_exists_locally
```

THE PATH ENVIRONMENT VARIABLE

- By default, `PATH` holds at least the relevant paths for your *system commands* (e.g. `/usr/bin`). It is a good idea to extend it in your `.bashrc` as follows:

```
PATH=$HOME/bin:$HOME/.local/bin:$PATH
export PATH
```

- Each path specified in `PATH` must be *delimited* by a colon "`:`". This is true for any environment variable that holds a list of paths, e.g. `MANPATH`, `INFOPATH`, `PYTHONPATH` and others.
- The `export` keyword ensures that your customized `PATH` is available in other shell instances as well
- `$HOME/bin`: a standard path in your home directory for executable scripts or programs
- `$HOME/.local/bin`: default path used by `python` to install packages in a Linux user directory (some of those packages come with executables and you want to access them). ***Example:*** the command below installs a Python package "`package_name`" below your `$HOME/.local` path by default:

```
$ python -m pip install --user <package_name>
```

THE PATH ENVIRONMENT VARIABLE

```
1 PATH=$HOME/bin:$HOME/.local/bin:$PATH  # note the last $PATH expansion!
2 export PATH
```

Order is important:

- You must dereference the content of `PATH` in your re-assignment of `PATH` (see line 1 above) in order to keep what was previously defined in it.
- *Append it at the end* to ensure that your custom executables (with possibly the same names as already existing ones) ***take precedence!***
- Once the shell has found a matching executable in `PATH`, it will not look any further.

Example: Assume you have the executables `$HOME/bin/my_exec` and `$HOME/.local/bin/my_exec` and your PATH is set as shown above. If you run

```
$ my_exec
```

the executable in `$HOME/bin/my_exec` will be executed.

SETTING SHELL VARIABLES

- You can omit the `export` keyword. In that case the variable will *only* be available in the current shell instance:

```
$ my_var='Hello World!'  # set a variable in the current shell (no spaces between '=')
$ bash                   # create another sub-shell instance
$ echo $my_var           # my_var is empty
```

- With `export`:

```
$ export my_var='Hello World!'  # set a variable in the current shell
$ bash                          # create another sub-shell instance
$ echo $my_var                  # value of my_var is propagated
Hello World!
```

> You must use `export` in your `.bashrc` or `.zshrc` files to ensure the settings propagate correctly.

- You can delete any variable using the `unset` command:

```
$ my_var='Hello World!'  # set a variable in the current shell
$ unset my_var           # unset it again in the current shell
$ echo $my_var           # my_var is empty
```

WHAT IS SHELL SCRIPTING?

- Typing out a series of commands that do complex tasks is not convenient and will only remain in your `history` for a short time.
- Shell scripting (and also Python scripts) is a powerful way to perform all kinds of *automation tasks*, often repetitive in time.
- By setting shell variables from the previous slides, you have already seen an important mechanism of shell scripting.

> A shell script is an *executable file* that contains commands together with pipes, file redirection and structures such as loops or if-conditionals to perform (more complex) tasks in the command line. A shell script allows you to archive and *replay* the commands in the script. Such (user) scripts are often placed in `$HOME/bin` and have permissions `700` (owner only) or `755` (group and others can read and execute as well).

SHELL SCRIPTING INTERPRETER

You should be specific about which shell (interpreter) you want to target in your scripts. This ensures *portability* of your scripts.

- You specify the interpreter with a *shebang*. The general form is:

```
#!interpreter_command [optional arguments]
```

  which you must write *at the very beginning* of your script.
- ***Examples are:***

bash:	`#!/usr/bin/env bash`
zsh:	`#!/usr/bin/env zsh`
python:	`#!/usr/bin/env python3`

Note: Use the `/usr/bin/env` command to resolve the actual path of the interpreter you target. Some users might have custom installations for these interpreters in their `PATH`. Hard-coding a path like `/bin/bash`, for example, would ignore this customization and possibly annoy users of your scripts.

SHELL SCRIPTING INTERPRETER

Your script ***must be executable***, like any other program. By now you know how to do that. The following is an example script called `my_exec` (see lecture code):

```bash
#!/usr/bin/env bash
echo "I am script $0, running inside $PWD."
echo "The following arguments were given:"
for arg in "$@"; do
    echo $arg
done
```

- Save the script inside `$HOME/bin` because we have setup this path is in `PATH`. The suffix `.sh` is optional, you can choose any name you want. It is just a file.
- Set executable permissions and run it:

```
$ chmod 755 ~/bin/my_exec
$ pwd
/home/fabs
$ my_exec Hello World!
```

- ***What output do you expect?***

SHELL SCRIPTING SPECIAL VARIABLES

There are some special variables that you can make use of in your scripts and functions:

`$@`	Expands to quoted arguments. *For previous example:* `"Hello" "World!"`
`$0`	The full path of the script. *(Always use `$0` for your help messages in case you rename your script later.)*
`$1`, ..., `$9`	The first *nine* script arguments. *For previous example:* `$1` → `Hello` and `$2` → `World!`
`$#`	The number of arguments passed to the script. *For previous example:* `$#` → `2`

A NOTE ABOUT STRINGS

Strings are very useful in scripts. They exist in two variants:

1. *Hard-quoted strings:* single-quotes

```
expansion=1234
str='This is a literal string, no variable ${expansion}' # single-quotes
echo ${str}
```

```
This is a literal string, no variable ${expansion}
```

2. *Soft-quoted strings:* double-quotes

```
variables='random values from other variables'
str="This string allows me to expand ${variables}" # double-quotes
echo ${str}
```

```
This string allows me to expand random values from other variables
```

SHELL SCRIPTING: `for`-LOOPS

Often you need to loop over a list of items obtained from another command invocation:

```bash
#!/usr/bin/env bash
dir=$1  # what does this line do?
for f in $(find $dir -maxdepth 1 -type f -name "*.py"); do
    # f:  iteration variable
    # in: expects a list of items (for iteration)
    echo $f  # you would do something more meaningful here
done
```

The "`$(...)`" executes the statement inside the parenthesis in a sub-shell and returns the `stdout` to the caller. You can use pipes for the commands in parenthesis as well. *Such command substitutions are very useful in shell scripts and have numerous applications.* ***Note:*** in the older Bourne shell, command substitutions were accomplished with backticks instead "`` `...` ``".

SHELL SCRIPTING: `if`-CONDITIONALS

The general form for an `if`-conditional looks like this:

```
if [ condition_A ]; then
    # execute this block when condition_A is true
elif [ condition_B ]; then
    # execute this block when condition_B is true
else
    # execute this block otherwise
fi # except for loops, the end-delimiter of constructs is the construct name in
   # reverse
```

Main reference for `if`-conditionals:

https://tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_01.html

if-CONDITIONALS: STRING COMPARISONS

String comparison condition:

```
if [ condition ]; then
    # code if condition is true
fi
```

`[ STRING1 == STRING2 ]`	Test for *equality*. For strict POSIX compliance you may use "=" instead of "=="
`[ STRING1 != STRING2 ]`	Test for *not* equal
`[ -z STRING ]`	True if the *length* of the string is zero
`[ -n STRING ]` or `[ STRING ]`	True if the length of the string is *non-zero*

Examples: (see lecture codes)

```
if [ 'abc' == 'abc' ]; then
    echo "'abc' == 'abc'"
fi

str='' # empty string
if [ -z $str ]; then
    echo 'str is empty'
fi
```

`if`-CONDITIONALS: STRING COMPARISONS

Example: compare a string argument

```
#!/usr/bin/env bash
if [ "$1" == 'Hello CS107/AC207!' ]; then
    echo 'Success!'
else
    echo 'Got unexpected string argument'
fi
```

What output do you expect from the following invocation?

```
$ ./string_comparison.sh Hello CS107/AC207!
```

`if`-CONDITIONALS: INTEGER COMPARISONS

Integer comparisons: the general form is

`[ INT1 OP INT2 ]`

where *OP* is one of the following:

`-eq`	`INT1` is *equal* to `INT2`
`-ne`	`INT1` is *not equal* to `INT2`
`-lt`	`INT1` is *less than* `INT2`
`-le`	`INT1` is *less than or equal* to `INT2`
`-gt`	`INT1` is *greater than* `INT2`
`-ge`	`INT1` is *greater than or equal* to `INT2`

See https://tldp.org/LDP/abs/html/comparison-ops.html

`if`-CONDITIONALS: INTEGER COMPARISONS

Example: integer comparisons

```bash
#!/usr/bin/env bash
if [ $# -gt 2 ]; then
    echo "Number of arguments $# is larger than two"
else
    echo "Number of arguments $# is less than or equal to two"
fi
```

Testing with different number of arguments:

```
$ ./int_comparison.sh a b
Number of arguments 2 is less than or equal to two
$ ./int_comparison.sh a b c
Number of arguments 3 is greater than two
```

`if`-CONDITIONALS: FILES AND DIRECTORIES

Often you need to test if files exist:

`[ -d FILE ]`	True if `FILE` exists and is a *directory*
`[ -f FILE ]`	True if `FILE` exists and is a *regular file*
`[ -e FILE ]`	True if `FILE` exists
`[ -r FILE ]`	True if `FILE` exists and is *readable*
`[ -w FILE ]`	True if `FILE` exists and is *writable*
`[ -x FILE ]`	True if `FILE` exists and is *executable*

Note that instead of `FILE` (which is some path to a file) you can also specify a *file descriptor* using `/dev/fd/n` with `n` the file descriptor ID. (***recall:*** `stdin`=0, `stdout`=1, `stderr`=2, ...)

See: https://tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_01.html

`if`-CONDITIONALS: FILES AND DIRECTORIES

Example: simple testing for files (`file_check.sh`)

```bash
#!/usr/bin/env bash
if [ $# -ne 1 ]; then
    cat <<EOF
USAGE: $0 <path/to/file>

    More documentation here.  The form used here is called a here-document.
    They are very useful to write longer strings and expanding variables like
    \$0 above.  See https://tldp.org/LDP/abs/html/here-docs.html
EOF
    exit 1 # exit with failure code
fi

if [ -f $1 ]; then
    echo "File $1 exists and is a regular file"
elif [ -d $1 ]; then
    echo "File $1 exists and is a directory"
elif [ -e $1 ]; then
    echo "File $1 exists and is an unknown file"
fi
```

You can find this example script in the lecture codes:
https://code.harvard.edu/CS107/main/tree/master/lecture/code/lecture03

SHELL SCRIPTING REFERENCE

- This the essential reference for `bash` scripting: https://tldp.org/LDP/Bash-Beginners-Guide/html/index.html
- More advanced topics: https://tldp.org/LDP/abs/html/index.html
- Writing scripts requires ***practice***.
- When you notice that you keep repeating a task over and over, write a script instead and save it in `~/bin` for example.
- `bash` scripts are extremely useful to automate tasks that involve *batch processing*. This may include filtering noise from data, generating movie frames, running periodic data backups or automating the submission of computing jobs on supercomputers.
- `bash` scripts are not very well suited for floating point arithmetic (use `python` for this).

JOB/PROCESS MANAGEMENT

- Any process or job is assigned a unique `PID`:

```
$ sleep 100
```

```
$ ps aux | grep sleep # this is run in another bash instance, you see why in a second
fabs      145691  0.0  0.0   5364   688 pts/4    S+   12:19   0:00 sleep 100
```

  The `PID` for this `sleep` process is `145691`

- The shell gives you some tools to manage such processes:
  - Run them in the background
  - Move a job to the foreground
  - Suspend a job
  - Terminate a job
- Running a process will ***block*** your prompt. The above command `sleep 100` will return back control only after 100 seconds have passed.

RUNNING JOBS IN BACKGROUND

- The shell returns control immediately by appending a "&"

```
$ sleep 100 &
[1] 153514  # the shell notifies us that PID 153514 is running in background
$ jobs      # we check that the job is running indeed
[1]+  Running                 sleep 100 &
...         # 100 seconds later the shell will tell you that the process has concluded
[1]+  Done                    sleep 100
```

- Appending a & will put the job in the *background*, you could exit the shell and the job would continue to run. (Only if the shell you are quitting is a *non-login shell*!)

SUSPENDING JOBS

- You can *suspend* a job to get back control of the shell by pressing `Ctrl-z`.

```
$ sleep 100
^Z  # pressed Ctrl-z here
[1]+  Stopped                 sleep 100
$ jobs
[1]+  Stopped                 sleep 100
```

  A stopped (or suspended) job *does not make progress*! If you want to quit the current shell (even if inside an interactive non-login shell) it will warn you the first time you try:

```
$ exit
exit
There are stopped jobs.
$ exit  # the second time you call exit (or Ctrl-d) the shell will quit without warning
```

- You can bring a stopped job back to foreground by using the `fg` command.

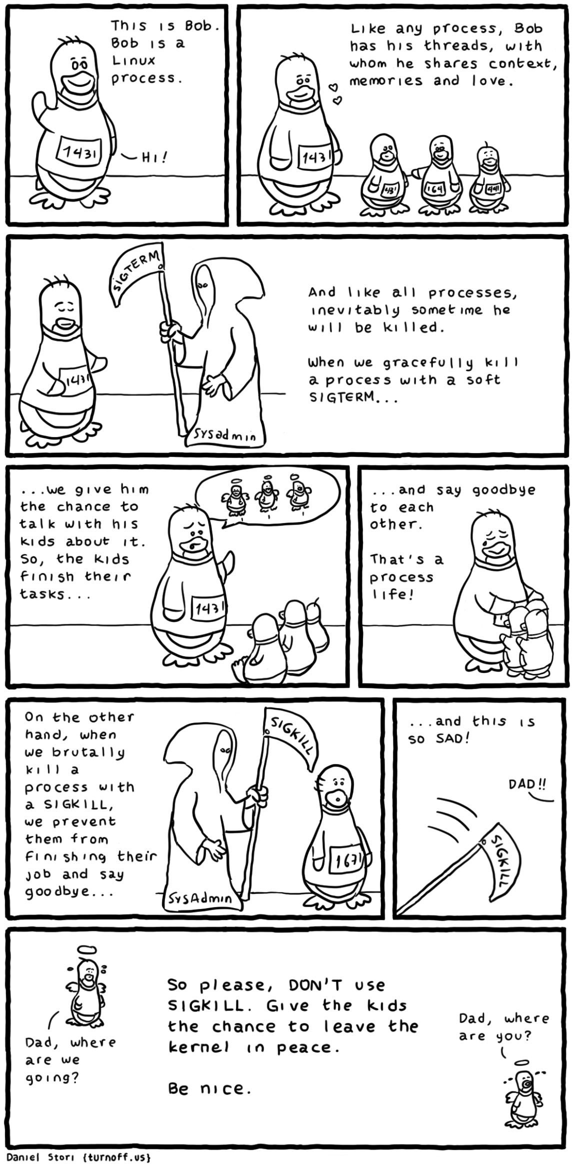
LIST JOBS AND PROCESSES

You can list and display running jobs and processes in several ways:

- `jobs` (see `man jobs`): displays the status of jobs in the current session (running, stopped, terminated)
- `ps aux` (see `man ps`): list all running processes on `stdout`. You may need to filter through `grep` to find what you are looking for.
- `top` (always available on Linux, see `man top`) to list a job ranking based on resource usage.
- Similar to `top` with better multi-core support: `htop` (see `man htop`). You may need to install it. In Ubuntu: `sudo apt-get install htop`
- Many others exists with more fancy presentation of data. For example https://github.com/cjbassi/ytop

TERMINATE JOBS

- You can terminate any job you have appropriate permissions.
- You can graciously terminate a job or *forcefully* kill it. (What would you prefer? 😅)
- You should only use the latter when there is no hope. ***Example:*** system starts to become unresponsive due to memory leak.
- The former will make sure that claimed resources are freed correctly and child processes are shutdown first. Relevant for multi-threaded programs.



TERMINATE JOBS

- You terminate jobs by sending a *signal* through the `kill` command. See:

```
$ whatis signal
signal (7)           - overview of signals
signal (2)           - ANSI C signal handling
signal (3p)          - signal management
$ man 7 signal
$ man kill
```

- By default `kill` will send a `SIGTERM` signal which is *graceful*. The rude one is called `SIGKILL`.
- You can specify the signal with the `-s` option of the `kill` command (be sure you get the `PID` right! Use `ps` or `top` to get it). ***Example:***

```
$ kill -s SIGKILL <PID>  # only do this when nothing else works anymore
```

- If you are sure that, for example, `python` is causing you trouble, you can send a `SIGTERM` by name using the `killall` command:

```
$ killall python
```

TERMINATE JOBS

- You terminate jobs by sending a *signal* through the `kill` command. See:

```
$ whatis signal
signal (7)           - overview of signals
signal (2)           - ANSI C signal handling
signal (3p)          - signal management
$ man 7 signal
$ man kill
```

- You can send an *interrupt* signal (`SIGINT`) by pressing `Ctrl-c`
- A `SIGINT` can be *catched* and processed differently by interactive software. For example, a hanging Python script will not always terminate with `Ctrl-c` because the interpreter will catch the signal and decide what to do with it. Use `killall python` instead.
- In most of the cases a `SIGINT` translates to `SIGTERM`

RECAP

- Take advantage of the shell customization and adapt it to best suit your workflow.
- I/O redirection and pipes are powerful tools that you must internalize when you spend the majority of time in the shell
- Process management and suspension is important once you are in the role of system administration or if you need more fine grained control over your processes.
- Environment variables allow you to further customize your system. They are very powerful and can be (and should be) used in shell scripting

Further reading:

- Bash beginners guide: https://tldp.org/LDP/Bash-Beginners-Guide/html/index.html
- Advanced Bash scripting: https://tldp.org/LDP/abs/html/index.html
- Bash startup files: https://www.gnu.org/software/bash/manual/html_node/Bash-Startup-Files.html
- S. R. Bourne, *"An Introduction to the UNIX Shell"*, Bell Laboratories (`reading/bourne1997.pdf` in main repository)
- S. R. Bourne, *"The Unix Shell"*, Byte Magazine, 1983. Link