Essential Linux

Junfan Zhu

junfanz@gatech.edu

Last update: 2022-12-10

Content

`README`: This document probably covers 99% practical developing and debugging commands on Linux, and saves you from searching everytime for those frequent usage that you may forget. To compose this document is also a process of my comprehension and skills enhancement. Feedbacks and corrections are welcome!

---

1. ls, cd, cp, cat, mv, mkdir, rm, touch

Syntax	Description
1. `find a`	see directory structure under `a`
2. `balabala Control + c`	skip whole line (if you mistype)
3. `balabala Control + u`	clear whole line (if you mistype)
4. `history`	all history you've typed
5. `cat tmp.cpp`	see the content of tmp.cpp
6. `Control + Insert`	copy a line in command line (to paste outside)
7. `Shift + Insert`	paste into command line (from outside)
8. `ls ../`	`..` goes to parent path
9. `ls ../tmp/./`	`.` current tmp folder
10. `ls ~/tmp/main.cpp`	`~` home path
11. `ls -l`	list long detailed info
12. `ls -hl`	same as `ls -l`, but change file size info to a humanized way: how many K
13. `ls -a`	list all file under path including hidden files, whose name starts with `.`
14. `ll`	same as `ls -a`
15. `ls -A`	same as `ls -a`, but not showing current path
16. `pwd`	show current path
17. `cd a`	change directory to `~/a` folder
18. `cd -`	back to previous directory
19. `mkdir a`	create folder `a`
20. `rm c -r`	delete folder `c`
21. `rm t.txt`	delete `t.txt`
22. `rm a/*`	delete everything inside `a` folder, but keep `a` folder
23. `rm * -r`	delete files in folder, but keep hidden files
24. `rm /* -rf`	go die, delete everything and run
25. `rm t.txt s.txt`	delete two txt files
26. `touch tmp.txt`	create file `tmp.txt`. (touch is create)

Syntax	Description
27. `cp a/tmp.txt b`	copy tmp.txt in `a` folder, paste into `b` folder
28. `mv a/tmp.txt b/`	move it from `a` into `b` folder
29. `cp a/tmp.txt b/tmp2.txt`	copy tmp.txt in `a` folder, paste into `b` folder, rename file
30. `mv a/tmp.txt b/tmp2.txt`	move it and rename it
31. `mv a/* b/`	move everything in `a` folder into `b` folder
32. `mv tmp.txt tmp2.txt`	rename `tmp.txt` to `tmp2.txt`
33. `cp a b -r`	copy folder `a` into `b`
34. `mkdir --h`	see all operations of `mkdir`
35. `mkdir a b c`	create folder `a`, `b` and `c`
36. `mkdir a/b/c -p`	make directory `a/b/c` one inside one

---

2. `tmux`, `vim`

2.1. `tmux`

- Split screen.
- Develop in `tmux`, there's no worry about data loss caused by interruption or network down, especially useful for long running code.
- Can open many sessions. Each session has many windows. Each window has many panes. Each pane has a shell interactive, can write codes and see outputs there.
- If no pane in a window, a window automatically disappear. If no window in a session, a session automatically disappear.

Syntax	Description
1. `tmux`	go into tmux, a new terminal pops up
2. `Control + d`	close and exit current pane
3. use mouse	to click on a certain pane on the screen; to drag the segmentation line between panes to adjust size
4. `Control + a, %`	split current pane into 2 panes, left and right
5. `Control + a, "`	split current pane into 2 panes, up and down
6. `Control + a, z`	full screen mode, second time: cancel full screen mode

Syntax	Description
7. `Control + a, d`	detached from session, jump from pane out out out of session
8. `tmux a`	tmux attach, reopen previously detached session
9. `Control + a, s`	up/down to select which session, left/right to expand or goback
10. `Control + a, c`	create a new window in current session
11. `Control + a, w`	select other windows, deeper than `Control + a, s` (session level)
12. `Shift`	select certain text

2.2. `vim`: editor in command line

Syntax	Description
1. [**Edit file - Step 1**] `vim tmp.txt`	open this file or create new file tmp.txt.
2. [**Edit file - Step 2**] `i`	edit mode.
3. [**Edit file - Step 3**] `ESC`	escape from edit mode, and go back to default mode.
4. `:`, `/`, `?`	either of the three will go to command line mode in the bottom.
5. `:w`	save
6. `:w!`	save enforced
7. [**Edit file - Step 4**] `:wq<Enter>`	save and quit
8. `:q`	quit
9. `:q!`	quit enforced
10. `0`	go to beginning of this line
11. `$`	go to the end of this line
12. `G`	go to the final line
13. `:n`, `nG`	go to n-th line, e.g. `:10` goes to 10-th line.
14. gg, `1G`	go to first line
15. `n<Enter>`	go to next n-th line, e.g. `3<Enter>` goes to below 3-th line
16. `n<Space>`	go to next n-th letter, e.g. `3<Space>` goes to right 3-th letter
17. `/word`	find first `word` below current location, e.g., `/int`, find first `int` below.
18. `?word`	find first `word` above current location, e.g., `/int`, find first `int` above.

Syntax	Description
19. `n`	repeat previous search, e.g., `int` will seach each `int` one by one from top to bottom.
20. `N`	repeat previous search in reverse order, e.g., `int` will search each `int` one by one from bottom to top.
21. `:n1,n2s/word1/word2/g`	between `n1` number row and `n2` number row, find `word1` string, replace `word1` by `word2`.
22. `:1,$s/word1/word2/g`	replace all `word1` by `word2`.
23. `v`	select certain text. To cancel selection, do `ESC` * 2.
24. `:noh`	no highlight
25. `d`	delete selected text.
26. `dd`	delete whole row. (Actually it's cut, can be pasted somewhere else.)
27. `y`	copy selected text.
28. `yy`	copy whole row.
29. `p`	paste selected text in current location. If you select whole line, it will be pasted in next row.
30. `u`	revoke, similar to `Control + Z`.
31. `Control + r`	cancel revoke
32. `Control + insert`	copy the selected content
33. `Shift + insert`	paste the selected content
34. `Shift`	can select any lines you want
35. `Shift + >`	move right the selected text
36. `Shift + <`	move left the selected text
37. `:set nu`	show which number of line you're at
38. `:set nonu`	hide which number of line you're at. This is useful when you want to copy the code without copying the line number by hiding the number.
39. `ggd100G`	delete the first 100 lines, 100 can change to any other number. Can break down to `gg + d + 100G`.
40. `100Gdd`	delete the 100-th row
41. `Gp`	paste the deleted line to the end of file last line.
42. `100Gyy`	copy the 100-th row
43. `G100<Space>i<BackSpace>ESC`	go to the final line, the 100-th letter, edit mode, delete this letter, return.
44. `3G8<Space>i<BackSpace>ESC`	go to the 3rd line, the 8-th letter, edit mode, delete this letter, return.

Syntax	Description
45. `gg30`	go to first line, 30-th letter.
46. `16<Enter>55`	go to 16-th line, 55-th letter.
47. `11G14<Space>v13G5<Space>d`	delete letters from 11-th row 15-th letter (including this letter) to 13-th row 5-th letter (including this letter). `v` is edit mode.
48. `G$p`	paste the deleted content to the end of file without starting a new row. Can be break down to `G$`: go to the last letter, `p` paste.
49. `ggdG`	delete everything from first line to last line.
50. `gg=G`	format all, remove redundant indentations.
51. `:set paste`	paste from outside without automatic indentation. (By default, vim has redundant indentations).
52. `:set nopaste`	cancel paste mode, and reopen automatic indentation. (When you've pasted already and want to edit the code, you need this to reopen automatic indentation).
53. `Control + q`	kill current job (if vim is not responding).
54. `-`	It's a `<Space>` in vim, need to delete them when copy/paste code.
55. Exceptions handling	If a file is opened twice, it's a confilct. You can either close that file by `:q`, or delete the `main.cpp.swp` file by `rm .main.cpp.swp`.

---

3. Shell

- Communicating with operating system using command line.
- Command line can be viewed as executing each line of shell file.

3.1. Variables

Syntax	Description
1. `vim test.sh`	create `test.sh`
2. [**First Line in a shell file**] `#! /bin/bash`	write this in the first line inside test.sh file, must specify bash as interpreter, otherwise cannot execute.
3. `#! /usr/env/python`	same logic, to specify python as interpreter and tell operating system.
4. [**Second Line in a shell file**] `echo "Hello World!"`	Need to save the file everytime before exit.
5. [**Execution Method 1, First Step**] `chmod +x test.sh`	to grant access to this file as executable, color change from white to green (executable).
6. [**Execution Method 1, Final Step**] `./test.sh` or `~/test.sh`	execute shell file.
7. [**Execution Method 2**] `bash test.sh`	No space near `=` when defining a variable! `name='abc'` or `name=abc` or `name="abc"`, all works.
8. `$`: calculate the value	`echo $name` or `echo ${name}` will output `abc`. `echo ${name}def` will output `abcdef`.
9. `readonly name` or `declare -r name`	2 ways of declaring `name` read only. After declaration, will no longer permit to change the variable, e.g. `name=def`.
10. `unset name`	delete variable `name`. After that, when you `echo $name`, no output.
11. `export name` `declare -x name`	2 ways of changing local variable `name` to global variable. (Subprocess cannot access local variable, but can access global variable.)
12. `export name=abc`	[**Change global variable `name` to local variable, Step 1: Define global variable**]
13. `declare +x name`	[**Change global variable `name` to local variable, Step 2: Change global variable to local variable**]
14. `''` and `""`	`'balabala'` will output as it is. `"balabala"` can be executed or pick up variables.
15. `man echo`	see echo tutorial.
16. `echo '$name \"hh\"'`	output `$name \"hh\"`.
17. `echo "$name \"hh\""`	output `abc "hh"`.

Syntax	Description
18. `echo $?`	return previous command exit code (not stdout), `0` means normal exit (it's similar to C++ main function return 0, means normal exit), other values means error. This is **very useful** for debugging.
19. `echo $(command)`	return stdout of `command`. This is useful to show the output (it's similar to `C++ cout`).
20. `echo ${#name}`	output length of `name` = 3.
21. `echo ${name:0:3}`	output substring of `name` from 0 to 3.
22. Pass parameters to shell	`$1` is the first parameter, `$2` is the second parameter, etc.

Example

Must have the first line `#! /bin/bash` to specify bash grammar.

```
#! /bin/bash

echo "File name:"$0
echo "First parameter:"$1
echo "Second parameter:"$2
echo "Tenth parameter:"${10}
```

Execute above shell and pass each value into it.

```
./test.sh 1 2 10
```

3.2. Array, `expr`, `read`, `echo`, `test`, `[]`

- `expr` gets `stdout` result. In logical expression, If true, `stdout` = 1; If false, `stdout` = 0.
- `test` gets `exit code` result, 0 = true, nonzero = false.
- `expr` also output its result in `exit code`.
  - In logical expression, If true, `exit code = 0`; If false, `exit code = 1`.
  - In numerical or other expressions, `exit code = 0`.

Syntax	Description
1. `array=(1 a "b")`	Create array, same as `array[0]=1`, `array[1]=a`, `array[2]="b"`.
2. `echo ${array[index]}`	read `[index]` element of an `array`.
3. `echo ${array[@]}` or `echo ${array[*]}`	Read whole array.

Syntax	Description
4. `echo ${#array[@]}` or `echo ${#array[*]}`	Actual length of array.
5. `echo $(expr length "$str")`	Output length of the str. Note: The `""` in `"$str"` is to avoid `<space>` in `str` (Hello`<Space>`World!) that causes error.
6. `echo $(expr index "$str" balabala)`	Output the index (start from 1) of any letter in `str` that has the same letter in `balabala`, if no such letter then return 0.
7. `echo $(expr substr "$str" 2 3)`	return a substr of `str` that starts from letter `2` and substr length = `3`
8. `echo $(expr $a + $b)`	return a + b
9. `echo $(expr $a - $b)`	return a - b
10. `echo $(expr $a \* $b)`	return a * b
11. `echo $(expr $a / $b)`	return a / b
12. `echo $(expr $a % $b)`	return a % b
13. `echo $(expr \( $a + 1 \) \* \( $b + 1 \))`	return (a + 1) * (b + 1). Note that each item in `expr` needs a `<Space>`, including `(` and `)` which also needs `\` ahead.
14. `echo $(expr $a '<=' $b)`, `echo $(expr $a \<\= $b)`	return comparison boolean 0 or 1
15. `echo $(expr 3 + 4 + 5)`	return 12
16. `a = 3, a = expr $a + 4, echo $a`	return 7
17. `read name, echo Hello, $name`	You enter `World`, it outputs `Hello, World`
18. `read -p "balabala" -t 5 name, echo Hello, $name`	It says `balabala` and suggest you to input, You enter `World`, it outputs `Hello, World`. It wait for your input for only `5` seconds, if you don't answer, after `5` seconds it ignores such command, and continues to the next command, which is `echo Hello, $name`, and output `Hello,`.
19. `echo "Hello World" > output.txt`	output into file `output.txt`. You can see the output by `ls, cat output.txt`.
20. `test -e test.sh && echo "exist" \\|\\| echo "Not exist"`	output exist if `test.sh` exist, vice versa. Note that after every `test`, need to enter another line `echo $?` to see its result, result `0` means true, `nonzero` means false.

Syntax	Description
21. `test $a -eq $b`	if `a` equals to `b`. Note that `eq` is equal, `ne` is not equal, `gt` is >, `lt` is <, `ge` is >=, `le` is <=.
22. `[ 2 -lt 3 ]`	True, return 0. Note that `[]` is same as `test` but more in if conditions. Everything inside `[]` needs `<Space>`.
23. `name="abc", [ "$name" == "abc" ]`	True. Note that every string including variable needs `""`.

3.3. If, Loop and Function

Syntax	Description
1. `if condition then balabala elif condition then balabala elif condition then balabala else balabala fi`	`fi` means end of if-condition (similar logic we have `case...esac`). It judges the **exit code** of the condition.
2. `for var in var1 var2 var3 do balabala done`	for loop. If infinite loop, `Control + C`.
3. `seq 1 10`	return 1 to 10 in each line, this can be written in command line terminal.
4. `for i in {1..10}`, `for i in {a..z}`	Note this cannot be written in command line terminal, but can use in for loop.
5. `for ((i=1; i<=10; i++)) do balabala done`	Needs two `(())`.
6. `while read name do echo $name done`	Read each variable `name` in the file and output one by one until the end of file, if condition is true, I continue. Note that when `read` comes to the end of file (`Control + d`), it returns `exit code` = 1.
7. `until condition do balabala done`	If condition is false, I continue. `until do` is the opposite of `while do`.
8. `break`	break from current loop, jump out from this loop layer. It's different from `C++` that: in Shell, `break` cannot jump out of `case`.
9. `continue`	jump out from current loop, for this time.

Syntax	Description
10. `func() {echo "balabala" return 123}  output = $(func) ret = $?  echo "output = $output"  echo "return = $ret"`	Output = balabala, return = 123. Note that call the function: `func`, not `func()`.

Example

```
func() { # calculate 1 + 2 + 3 + ... + `$1` by recursion
  if [ $1 -le 0 ]
  then
    echo 0
    return 0
  fi

  sum=$(func $(expr($1 - 1))
  echo $(expr $sum + $1)
}

echo $(func 10) # output = 55
# sum is calling func(9), and then echo sum + 10.
```

Note

- `$0` inside the function is the filename, not function name.
- `$1` is the first parameter, `$2` the second parameter, etc.

3.4. Files

How to copy out the files in server

- Exit `tmux`, `cat filename` to show content in the file.
- Use mouse to select text beginning, drag the mouse to the end of the file.
- `Shift`, and use mouse to click on the end of the file, it will select everything in the file.
- In Windows/Linux, `Control + Insert` to copy all; In Mac, `Command + c` to copy all.

Syntax	Description
1. `command > file`	redirect `stdout` into `file`, override `file`, e.g. `ls > output.txt`
2. `command >> file`	redirect `stdout` into `file`, add new content to current `file`
3. `command < file`	redirect `stdin` into `file`
4. `. filename` or `source filename`	This imports whole `filename` into current file

4. ssh

Syntax	Description
1. `ssh user@hostname`	Connect to remote server.
2. `~/.ssh/config`	Configuration, make a short nickname for long IP address. Go to ssh folder (`mkdir .ssh`), create a file `vim config`, after the above code block setup, you can connect to server next time by `ssh myserver`.
3. `ssh-keygen`	Generate public and private rsa key pair.
4. `ssh-copy-id myserver`	To connect to server without a password. You can also copy the public key (`cat id_rsa.pub`) to `~/.ssh/authorized_keys` file (go to `ssh myserver`, put password, `cd .ssh/`, `vim authorized_keys`, paste public key). Next time you `ssh myserver`, it will be automatic connection, no password needed.
5. `ssh myserver ls -a`	Show all files on server
6. `ssh myserver 'for (( i = 0; i < 10; i ++ )) do echo $i; done'`	Run command on server. Note that use `''` to quote on command with variables to avoid error. SSH will redirect the output in server to our terminal.
7. `scp -P 22 source1 source2 destination`	Copy file `source1` and `source2` (multiple files) to `destination`, specify server port `22`, note that this is Capital P.
8. `scp source destination`	`source` can be a file on website, copy this file to `destination`.
9. `scp -r dir/ myserver:/home/abc`	Copy/Upload whole local folder `dir/` to `myserver` under `:/home/abc` directory. This can upload a batch of files, no need to upload single file one by one. Note that here `-r` is in the front, not at the end.
10. `scp -r ~/tmp myserver:/home/abc`	Same thing. Copy/Upload a local file `tmp` to `myserver` under `:/home/abc` directory.
11. `scp -r myserver:tmp/a.txt .`	Copy `~/tmp/a.txt` file on `myserver` to local current path.

ssh config

```
# ~/.ssh/config

Host myserver1
  HostName: # IP address
  User: # user name
Host myserver2
  HostName: # IP address
  User: # user name
```

---

5. Git

5.1. Add, Commit, Roll Back

The logic of `git` is the same as a linked list for each branch, adding a new node and move current `HEAD` to the next node. There're many branches and forms a tree structure.

- Working Directory $\rightarrowtail$ Stage (Index) $\rightarrowtail$ Repository

Syntax	Description
1. `git config --global user.name abc`, `git config --global user.email abc@abc.edu`	Set up your user name as `abc`, emial as `abc@abc.edu`. This info is stored in `.gitconfig` file.
2. `git init`	Initialize current folder as git reposiroty. Hidden repo info is in hidden folder `.git`.
3. `git status`	If you modify a file but haven't added, if you've added a file but haven't committed, it will tell you.
4. `git add tmp.txt`	Add this file to stage, but you haven't committed yet.
5. `git add .`	Add all modified files to stage.
6. `git commit -m "balabala"`	Commit this file with notes `"balabala"`
7. `git diff tmp.txt`	Compare difference between current file in stage and its historical version in working directory, before you commit.
8. `git log`	See all historical versions under this branch.

Syntax	Description
9. `git reflog`	See the history of `HEAD` version movements, including rolled-back versions. Because you won't see the version in `git log` if you've deleted the most recent version.
10. `git checkout - tmp.txt` or `git restore tmp.txt`	Discard changes in working directory (that haven't been added to Stage) of file `tmp.txt`. Note that `git restore` didn't roll back to previous version, it just delete the version in working directory and roll back to the version in Stage.
11. `git restore --staged tmp.txt`	Unstage `tmp.txt`, just remove the file from Stage, *but you still need this file.*
12. `git rm -- cached tmp.txt`	Remove `tmp.txt` from cache, and you *no longer need this file.* If you still need it, do `git add tmp.txt` again, then it's no difference from unstage command above.
13. `git reset --hard HEAD^` or `git reset --hard HEAD~`	**Roll back** current version to previous version.
14. `git reset --hard HEAD^^`	Roll back current version to previous second version (two versions ahead).
15. `git reset --hard HEAD~100`; `git reset --hard abcdef`	Roll back current version to previous 100-th version (100 versions ahead); Roll back or forth to version `abcdef` (first 6-letter of a version number).

5.2. Push, Pull, Branch, Stash

Syntax	Description
1. `git branch`	`*` shows which branch you're at.
2. `git branch branch_name`	Create new branch.
3. `git checkout -b branch_name`	Create and switch to branch `branch_name`.
4. `git checkout branch_name`	Switch to branch `branch_name`.

Syntax	Description
5. `git merge branch_name`	Merge `branch_name` to current branch. If automatic merge failed, probably both modifications are done on the file, then you need to fix conflicts and then commit. After that, you can delete one branch with conflict.
6. `git branch -d branch_name`	Delete local branch `branch_name`.
7. `git push origin branch_name`; `git push -u`	Push local `branch_name` branch to remote repository; Push current branch to remote repository. The first time you need `-u`, in future no need.
8. `git push --set-upstream origin branch_name`	Create local branch `branch_name` to remote repository branch `branch_name`. Note that the current branch `branch_name` has no upstream branch, so you need to push the current branch and set the remote as upstream. This may happen when you switch to another branch but this branch doesn't exist on remote repository yet.
9. `git push -d origin branch_name`	Delete remote repository branch `branch_name`.
10. `git pull` or `git pull origin branch_name`	Merge current branch with remote repository branch, current branch or `branch_name` branch. Pull = Download + Merge.
11. `git branch --set-upstream-to=origin/branch_name1 branch_name2`	Local branch `branch_name1` set up to track remote `branch_name2`.
12. `git stash`	Use **stack** to store any modifications in working directory or in stage, that haven't been commited.
13. `git stash list`	See whole elements in the stack of `git stash`.
14. `git stash apply`	Revert this modification back to current branch, *but not delete this element.*
15. `git stash drop`	Delete the stored modification on the top of the stack.

Syntax	Description
16. `git stash pop`	Pop out the first element in the stack, it means revert this modification back to current branch, *and delete this element.* (You may need to handle conflict by popping out again on the main branch when you pop out on a second branch and return to the main branch while the main branch has already been modified. To avoid this, you'd better commit every time before you switch to a different branch.)

---

6. `thrift`: multiple servers communication

6.1. `thrift`

- `thrift` is an **RPC (Remote Procedure Call) framework**, which can call any functions on any server from any server. This enables us to write `Python` on a server that calls another `C++` API on another server, by receiving commands and passing results.
- `thrift` server needs multi-threading. Multiple times of calling APIs will generate a lot of threads, if you push the message queue from all threads at the same time, it will have bug (e.g., queue head override inconsistency issue).
- `Apache thrift` can be used in Micro-services. Tutorial: https://thrift.apache.org/tutorial/cpp.html
- `thrift` is faster than `socket`, you can focus more on business side with this tool, without reinvent the wheel. Wtih `socket` you need to define serialization and implement message communication interface between `cpp` and `python`, manage the `socket`, which is much more code.

Technical Notes

- `./main`: After compilation, launch server.
- `python3 client.py`: Execute `client.py`.
- If you have compiled 100 files together, then you only need to compile the files that are modified, no need to compile all files again.
- `make` identifies which files are modified, it only operates on the modified files. But we not often use `make` because one man can only write a few `cpp` files, not 100 files, so `g++ -c main.cpp` is enough.

Syntax	Description
1. `thrift -r --gen cpp ../../thrift/match.thrift`	After creating the interface, and `mkdir src` to create source file folder, in this folder we generate the server in `cpp`, where the thrift path is `../../thrift/match.thrift`. (`thrift -r --gen py ../../thrift/match.thrift` for `python`) Then do `ls`, we can see `gen-cpp` folder, do `tree .` we can see the structure in this folder, there're many `cpp` files already been automatically created, among which `skeleton.cpp` is equivalent to the `main.cpp` we want.
2. `g++ -c main.cpp match_server/*.cpp`	[**`C++` compilation, Step 1: compile**] Compile `main.cpp`, and compile all `cpp` files in `match_server`. We only compile `cpp` file, no need to compile `h` file (header automatically involves in the compilation). You'll see many `.o` files which means they are compiled.
3. `g++ *.o -o main -lthrift`	[**`C++` compilation, Step 2: link**] Link all `.o` files together, using `-lthrift` dynamic-link library (DLL). After link, we can run `./main` and it will run. Use `git restore --stage *.o` to not include `main` executable file and `*.o` compiled files before you push to `git`, it's good habit to only commit source files.c Same, use `git restore --stage *.pyc` to not include `python` compiled intermediary file and `git restore --stage *.swp` to not include vim file in `git push`.
4. `g++ *.o -o main -lthrift -pthread`	If you compile `cpp` code which used thread, you need to add `-pthread` here.

6.2. Producer-Consumer Model

Producer-Consumer Model in `C++` by creating a thread. Between Consumer and Producer, is message queue `q`. In message queue we need locks, because sometimes Consumer and Producer both need to write or execute `q`, may causing conflicts at the same time, so we need to put this message queue inside two locks. `mutex` has 2 operations:

Thoughts

- Can I avoid using threading in consumer? What if I execute task once I receive a request?
  - No. Because when a new request come, there's no guarantee **when** there's a match. 2 logics (Receiving request and Matching) are independent from each other and not the same. Single threading cannot make it.
- Why lock is needed in multi-threading?
  - Because multi-**threads** shares the same one memory space. (e.g. Global variable `message_queue` is shared by thread `consumer` and thread `message_queue.push` when receiving request.) But each **process** has independent stack space.

6.3. Multi-Threading and Lock

Semaphore $\rightarrowtail$ Lock $\rightarrowtail$ Conditional Variable

The nature of a lock is **semaphore**. Mutex is a **mutual exculsive** semaphore. Semaphore has two atomic operations:

- `P(s)`: add lock. To obtain the lock and make sure other threads are blocked outside the same code and only one thread is occupying this queue
- `V(s)`: release lock. To release lock to other potential threads once this thread has finished its job.

`s` means how many operations can be done and share this thread at the same time. If `s > 0`, we execute `P(s)` or `V(s)`, and `s -= 1`. If `s = 0`, it's blocked, we won't restart execution until `s > 0`. If `s = 1`, it's mutual exculsion (mutex), only 1 thread can occupy the code at one time.

Condition variable is an encapsulation of lock.

6.4. *User Match System*

6.4.1. Architecture

Client - match_server - save_client_data

Server (Game) - match-client

thrift - Server $\rightarrowtail$ remove/add user $\rightarrowtail$ Client

```
# Create thrift
vim match.thrift
```

```
// In the file
namespace cpp match_service

struct User {
  1: i32 id,
  2: string name,
  3: i32 score
}

service Match {
  // Interfaces pass in user and other info, return int type
  i32 add_user(1: User user, 2: string info),
  i32 remove_user(1: User user, 2: string info)
}
```

6.4.2. C++ Demo

```
#include <thread>
#include <mutex>
#include <condition_variable>
#include <queue>
#include <vector>

struct Task
{
  User user;
  string type; // operation type
};

struct MessageQueue
{
  queue<Task> q; // a queue to store Task
  mutex m; // a lock
  condition_variable cv;
}message_queue;

// a pool to store all users' info and match users
class Pool
{
  public:
  // operations to add or remove user
    void save_result(int a, int b)
    {
      printf("match result: %d %d\n", a, b);
    }
    void match(User)
    {
      while (users.size() > 1)
      {
        auto a = users[0], b = users[1];
        // delete first user
        users.erase(users.begin());
        // then the second user become the first user, delete this user again
        users.erase(users.begin());
        save_result(a.id, b.id);

      }
    }

    void add(User)
    {
      users.push_back(user);
    }

    void remove(User)
    {
      // need to traverse user_id to remove user
      for (uint32_t i = 0; i < user.size(); i ++ )
        if (users[i].id == user.id)
        {
          users.erase(users.begin() + i);
          break;
        }
    }
  private:
  // user info
    vector<User> users;
}pool;

class MatchHandler : virtual public MatchIf {
  // This class is generated by `thrift -r --gen cpp ../../thrift/match.thrift`
  public:
    MatchHandler() {
      // init
    }

    int32_t add_user(const User& user, const std::string& info) {
      // implementation
      printf("add user\n");

      // add a lock `m` above push operation on message queue.
      unique_lock<mutex> lck(message_queue.m);
      message_queue.q.push({user, "add"});
      // There's no need to unlock.
      // Because there's a destructor to automatically unlock after the function is done.

      // Use condition variable to wake up this function
      // notifying all threads that are waiting. Wake up!
      message_queue.cv.notify_all();
      // notify_all() or notify_one() to notify random one.

      return 0;
    }

    int32_t remove_user(const User& user, const std::string& info) {
      // implementation
      printf("remove user\n");
      // same as above
      unique_lock<mutex> lck(message_queue.m);
      message_queue.q.push({user, "remove"});
      message_queue.cv.notify_all();

      return 0;
    }

}

void consumer()
{
// If no lock or multi-threading, there will be a bug!
// Because consumer will end up in infinite loop, never get executed.
  while (true)
  {
    // Pass in the lock `m`.
    unique_lock<mutex> lck(message_queue.m);
    if (message_queue.q.empty())
    {
      // If message queue empty, this thread is waiting, to avoid infinite loop.
      // Use condition variable to wait until it's notified_all() by other threads.
      message_queue.cv.wait(lck);
      continue;
    }
    else
    {
      // If message queue not empty, pick up the front, and delete it.
      auto task = message_queue.q.front();
      message_queue.q.pop();
      // Unlock immediately after finishing with sharable variable, before doing task
      // To reduce waiting time for other threads using message_queue.
      lck.unlock();
      // do task
      if (task.type == "add") pool.add(task.user);
      else if (task.type == "remove") pool.remove(task.user);

      pool.match();
    }
  }
}

int main() {
  // Thread 1: Consume. To keep executing an infinite loop.
  thread matching_thread(consumer);

  // Multi-threads
  // Whenever server receivex any request, it allocates a thread to execute function.
  server.serve();

}
```

---

7. Environmental Variable and Pipe

7.1. Environmental Variable

Environmental Variables	Description
1. `HOME`	home directory.
2. `PATH`	Path where all the commands are stored. When you write a command, Operating system will traverse all commands in `PATH` until it finds the **first** command that matches your command.
3. `LD_LIBRARY_PATH`	`C++` distinguish dynamic-link and static-link. If main function uses dynamic library, it will find the `.so` file with this function when execute.
4. `CPLUS_INCLUDE_PATH`	Path of `C++` `.h` header files.
5. `C_INCLUDE_PATH`	Path of `C` `.h` header files.
6. `PYTHONPATH`	Path of `python` imported packages.
7. `JAVA_HOME`	Path of `jdk`.
8. `CLASSPATH`	Path of `Java` imported classes.

Environmental Variables are global variables that can be visited by all processes. Linux uses environmental variables to log config info, and we can change system config by changing environmental variables.

Technical Notes

- You can store the path in `.bashrc`, next time you open the command line, you don't need to `cd path` every time, but can run functions directly.
- When you add path, add path on the top, because duplicated environmental variables will only be executed the first one in order.

Syntax	Description
1. `env`	Show current user's variables.
2. `set`	Show current `shell` variables, including current user's variables.
3. `export`	Declare environmental variables that are exported from user's variables.
4. `echo $PATH`	Output a value of environmental variable. (e.g. `echo $HOME`)
5. `export PATH=/home/a/b:$PATH`	Change current variable `PATH` to `/home/a/b` directory, and use `:$PATH` save current info of `PATH`. Note that `:` is because all paths are separated by `:` in `PATH`, `$` is to get values from `PATH`. Then we do `cd ~/`, it shows your home directory is now changed to `/home/a/b`.
6. `vim ~/.bashrc`, then add the command (e.g. `export HOME=/home/a/b`) to the last line, then `source .bashrc` to apply this change.	To apply the change of environmental variable to all environments in the future, we need to put the command in `~/.bashrc` file. Then `source .bashrc` to apply this change to current `bash` environment and all the future environments, because every time we start `bash`, it will run `~/.bashrc` file; every time we `ssh` remote server, or `tmux` a new `pane`, it will start `bash`.

7.2. Pipe

Pipe redirects previous command's `stdout` to next command's `stdin`. It's useful for batch processing on many files by connecting commands together.

- Pipe only redirects `stdout`, it ignores `stderr`.
- The next command on pipe's right must accept `stdin`.

- Many pipes can connect one by one.

Difference between pipe and file redirection:

- File redirect output to a file `echo "Hello" > output.txt`). Left is command, right is file.
- Pipe's left is command and `stdout`, right is command and `stdin`.

Pipe command

`find . -name '*.py' \| xargs cat \| wc -l`

- Calculate total rows of all the `.py` files under current directory.
  - **Notes:** `find .` means find in current directory, `\|` is a pipe to output to next command.
  - `xargs` turns `stdin` to file's content, using `<Space>` to separate contents in `stdin`, as an input and pass into `cat`.
  - `xargs cat` is to `cat` show contents of all the files, same as `cat ./a.py ./b.py ./c.py and so on`.
  - `wc` is total rows of `stdin`.
  - Without `xargs`, the output will only be total number **file names**, not the total rows of **contents** of the files.|

Notes on `xargs`

- After `find` command, we need `xargs` on the right of the pipe, because `find` returns file name, and `xargs` turns file name into command parameter.
- After `cat` command, we don't need `xargs`, because `cat` filters on the content.

7.3. Frequently-Used Linux Commands

7.3.1. Tools

Syntax	Description
1. `md5sum main.cpp`	Return `md5` hash value for file `main.cpp`. (Encryption for some APIs to connect to port, or match password without looking at the password by comparing the hash value only.)
2. `time command`	Execution time of `command`.
3. `ipython3`	Quick calculator. It's much better than `expr` in `shell` when you have many `()`.
4. `watch -n 0.1 command`	Execute `command` every `0.1` second.
5. `tar -zcvf xxx.tar.gz /path/*`	Compress all files under `/path` to `xxx.tar.gz`. (No `*` also works.)
6. `tar -zxvf xxx.tar.gz`	Unzip file.

Syntax	Description
7. `diff x y`	Find difference between `x` and `y` file. If not output, then `x`, `y` are the same.
8. `sudo command`	Execute `command` as `root` user, `root` is the king in operating system and can do anything.
9. `apt-get install xxx`	Install `xxx` software. If you don't have permission, add `sudo` in the front.
10. `pip install xxx --user --upgrade`	Install `python` packages.

7.3.2. System

Syntax	Description
1. `top`	Linux task manager, to show CPU processes info. Put `M` to rank by memory, put `P` to rank by CPU, put `q` to quit.
2. `df -h`	Show hard disk usage, `-h` means in a humanized way to show the size in G or M.
3. `free -h`	Show memory usage.
4. `du -sh`	Show current directory occupies how much space on hard disk.
5. `ps aux`	Show all processes.
6. `ps aux | grep abc`	Search for one specific process `abc`, `|` is pipe, `grep` is a string matching tool to find a name includes `abc`. It will show 2 processes, the first one being that process we want to find, the second one is current `grep` process. (After you find this process, you can kill it.)
7. `kill -9 pid`	To kill a Process ID = `pid`. If `pid = 6298`, then `kill -9 6298`. Note: `-9` means `SIGKILL` signal. If you want to use `SIGTERM` signal, do `kill -s SIGTERM pid`.
8. `netstat -nt`	Check Internet connection. This is useful if you want to check on `ssh myserver`, there may be many IP connections from Internet worldwide.
9. `w`	List current logging in user.

Syntax	Description
10. `ping www.google.com`	See if you can access Google, to check your laptop has Internet connection.

7.3.3. File Search

Syntax	Description
1. `find /path/ -name '*.py'`	Search all `.py` files under `path`.
2. `grep balabala`	Read data from `stdin`, filter on the content if it contains string `balabala`, output this line and highlight `balabala` in red.
3. `find path/ | sort`	Sort content of every line (file names under `path`) by alphabetical order.
4. `head -5 main.txt > top.txt`	Get first 5 rows in `main.txt`, store the result in `top.txt`. Same logic, `tail -5` means the last 5 rows.
5. `tree` or `tree /path/`	Show structure of current directory. `tree . -a` includes hidden files.
6. `find /path/ -name '*.py' | xargs cat | grep 'balabala'`	Search from whole folder and all files, which line contains string `balabala`, output this line with highlight. `xargs` read the `stdin` result from pipe, and put the file name as pass-in parameter to `cat`, `cat` list the **content** of file (If without `xargs` to convert `stdin` to pass-in parameter, `cat` will only list the file **name**). Note that we only know if there's a line that contains `balabala`, but we don't know this line belongs to which file. To do so, we need `ag balabala`.
7. `ag balabala`	Global search and show which file which line contains string `balabala`. This is much more intelligent than above command, very useful.
8. `wc main.cpp`, `wc match_server/*`	Count total lines, words and characters, can do either single file `main.cpp` or a bunch of files under a directory `match_server/*`.

Syntax	Description
9. `find . -name '*.cpp' | xargs cat | wc - l`	Count total code lines of all `.cpp` files under this directory. `find . -name`: find all the files under current directory. `wc -l`: lines, `wc -w`: words, `wc -c`: characters.
10. `echo $PATH | cut -d ':' -f 3,5`	Output `PATH`, and use `:` to cut the last 3 and 5 rows. Note that `3,5` means row `3` and row `5`; `3-5` means from row `3` to row `5`.
11. `echo $PATH | cut -c 3-5`	Output `PATH`'s character 3 to character 5.
12. `cat a.txt | cut -d ' ' -f 1 | sort > b.txt`	Pick (cut) every first (`1`) word in every row in `a.txt`, split by `' '` space, and sort in alphabetical order, store the result in `b.txt`. If you want second word in every row, change `1` to `2`.

7.3.4. File Permissions

10 digits file permission: `drwxrwxr-x`

- `d`: If this file is a folder or a hyperlink
- `rwx`: read + write + execute
- There're 3 pairs of `rwx`, it use `-` if it's none.
- First `rwx` applies to myself, second `rwx` applies to team member, third `rwx` applies to other users.

`chmod +x abc` - `+x` can be `+r`, `+w`, `+x`, to add permission to read, write, execute for file `abc`. - `chmod -x abc` or `-r`, `-w`: to remove permission for file `abc`.

---

8. Docker

8.1. Rent a Cloud Server

- [**1. Terminal and Central Server**]: like a rough apartment, can be customized, belongs to cloud platform, cannot transfer environment, can call other services.
  - Framework or libraries: `Django`, `thrift`
  - Use `ssh` in terminal as a window to connect to this server.
    * `ssh root@111.11.111.1`, copy your public IP address generated on Cloud Platform (We **must** have public IP address!). We'd better create another user `adduser zjf`, and assign user `zjf sudo`

        permission by `usermod -aG sudo zjf`, so as not to do damage with `root` identity.
        * `vim .ssh/config` to setup `HostName 111.11.111.1` and `User zjf`. `ssh-copy-id server1` for no-password log in.
    - Basic version: 1 core (1vCPU), 2GB memory.
    - Image: Ubuntu 20.04. (20.04 is required for Docker)
    - Config environment for rough server
        * `sudo apt-get update`, always update first
        * `sudo apt-get install tmux`, only need to install `tmux`, no need to store other things (those will be on Docker)
        * Setup `tmux` as default config: Go back, `scp .bashrc .vimrc .tmux.conf server1:`, then `ssh server1` again.
- [**2. Second Layer: Cloud Platform**] Services (like a hotel, can't be customized, just a tool)
    - `socket`: IP + port
    - `http`, `mysql`, `cdn`, `redis`
- [**3. Third Layer: Docker, our main dev place**]
    - Docker can open as many small "virtual servers" as possible, on current server you've rented. (Build houses inside a house.)
    - Docker is good at transferring, from Linux to Windows, from Google Cloud to AWS, etc.
    - With unified image provided by Docker, environment configuration is no longer needed.
    - Use `attach` to go from Central server to Docker container. Or, a better way is to use `ssh` to log in directly from terminal to Docker.

8.2. Docker Image and Container

- Docker has many images, image is like a template. Each image has many containers.
- Containers generated by the same image has the same environment.
- Image is a mold or **stamp**, container is a product or **pattern**. A stamp can generate many patterns, all are the same.
- Each container is an independent cloud server.
- If we want to transfer Docker image, we're transfering the container. Use container to generate an image, download image as a compressed file, then load this file on server in Docker, and use image to generate a new container.

8.2.1. Docker Image

Syntax	Description
1. `sudo usermod -aG docker $USER`	Add user to `docker` group (after `ssh server1`), to avoid each time in need of `sudo` in Docker.

Syntax	Description
2. `docker pull ubuntu:20.04`	Docker pull an image.
3. `docker images`	Show all local images.
4. `docker image rm ubuntu:20.04` or `docker rmi ubuntu:20.04`	Delete image `ubuntu:20.04`.
5. `docker [container] commit CONTAINER_IMAGE_NAME:TAG`	Create image for `container`. (`[container]` is optional, you can add string `container`, or delete this, both works.)
6. `docker save -o ubuntu:20.04.tar ubuntu20.04`	[**Transfer image Step 1**]: Load `ubuntu:20.04` image and save as local file `ubuntu:20.04.tar`. (You can add read permission for others by `chmod +r ubuntu:20.04.tar`)
7. `docker load -i ubuntu:20.04.tar`	Load image from local file `ubuntu:20.04.tar`.
8. `scp server1:ubuntu:20.04.tar .`, then `scp ubuntu:20.04.tar server2:`. Then `ssh server2`, and `docker load -i ubuntu:20.04.tar`.	[**Transfer image Step 2**]: Copy `ubuntu:20.04.tar` to local `server1` (e.g. Google Cloud server), and then upload `ubuntu:20.04.tar` to `server2` (e.g. AWS Cloud server). Then go to `server2` to pull this image. Finally you can see it by `docker image`.

8.2.2. Docker Container

Technical Notes

- `docker export/import` and `docker save/load`
  - `export/import` discard historical records and metadata, only keeps the snapshot of container at that time.
  - `save/load` save whole records, with bigger size.
  - When you do `export/import`, you're transferring the **image** of container, not the container itself. Because container is not transferrable, container can generate template, **container's template** is transferrable.

Syntax	Description
1. `docker create -it ubuntu:20.04`	Create a container, using image `ubuntu:20.04`.
2. `docker ps -a`	Show all containers. (`docker ps` without `-a`: show all containers that are currently running.)

Syntax	Description
3. `docker start container_id`	Start docker container, container ID = `container_id`. If you do `docker start container_name` it also works. You can see container ID in the first column of `docker ps -a`, container name is the last column.
4. `docker restart container_id`	Retart docker container, container ID = `container_id`.
5. `docker stop container_id`	Stop docker container, container ID = `container_id`.
6. `docker run -itd ubuntu:20.04`	Create and start a container. (Note that `itd`: if you only do `it` without `d`, it will create and start and enter this container, `d` means not enter container.)
7. `docker attach container_name`	Enter container. (When you enter a container, you are under root directory, this can be viewed as a new server or virtual machine.)
8. `Control + p`, then `Control + q`	Exit container without shutting down the container. (If you do `exit` you shut down the container and exit.)
9. `docker start container_name` and then `docker exec container_name command_balabala`.	Execute command `command_balabala` in container `container_name`, after you've started this container. (e.g. `docker exec zjf ls`)
10. `docker rm container_name`	Remove a container. `rm` deletes container, `rmi` deletes image. (Note that you can't delete a running container, before deletion you need to stop first.)
11. `docker container prune`	Remove all containers that are stopped. (Note that here `container` word can't be omitted)
12. `docker rename container_1 container2`	Rename container.
13. `docker update container_name --memory 500MB`	Set container memory.
14. `docker stats`	See CPU, memory, storage, network info of docker. `Control + c` to exit.

Syntax	Description
15. `docker export -o balabala.tar container_name`	Export container `container_name` to file `balabala.tar`. (Add read permission `chmod +r balabala.tar`. Load to local server `scp server1:balabala.tar .`, load from local server to another cloud server `scp balabala.tar server3:.`)
16. `docker import balabala.tar image_name:tag`	Import local file `balabala.tar` as image, name it as `image_name:tag`
17. `docker top container_name`	Show all processes in `container_name`, after you've done `attach`.
18. `docker cp balabala CONTAINER: bala` or `docker cp CONTAINER:balabala bala`	Copy file from container to local, or from local to container. (Note that `bala` can be either a file or a path, e.g. `/root .`. Docker copy path doesn't need `-r`.)

8.3. Demo: Deploy Docker to Cloud Server

Steps	Command
1. Go to terminal, upload image to your cloud server.	`scp balabala.tar server1:`
2. `ssh` connect your cloud server.	`ssh server1`
3. Load the image ("stamp") to local.	`docker load -i balabala.tar`
4. Check this image.	`docker images`
5. Create and run the image ("stamp the pattern") `balabala:1.0` on local. `-p` is to map container `port 22` to local `port 20000`, because local `port 22` is already used.	`docker run -p 20000:22 --name my_docker_server -itd balabala:1.0`
6. Enter docker.	`docker attach my_docker_server`
7. Set up password (root user cannot modify password, can only setup password).	`passwd`
8. Exit container without shutting down.	`Control + p, Control + q`
9. Login as root user. `localhost 111.11.1.111`, `port 20000`	`ssh root@localhost -p 20000`

Just now we built our own cloud server on Google cloud server. Purpose of our own isolated cloud server is to be able to transfer between Google and AWS (Actually this Matryoshka can go infinite if you want). Now we want to log in our own cloud server from everywhere.

Steps	Command
10. Setup user `zjf`.	`adduser zjf`
11. Grant `sudo` to user `zjf`.	`usermod -aG sudo zjf`, then `logout`
12. Log in as user `zjf`.	`ssh zjf@111.11.111.1 -p 20000`
13. Install `sudo` command (under root).	`apt-get install sudo`
14. Install `tmux`.	`sudo apt-get install tmux`
15. Change config to setup no-password login.	`vim .ssh/config`

16. Add these in config file

```
Host server1_docker
  HostName 111.11.111.1
  User zjf
  Port 20000
```

Steps	Command
17. Setup no-password login.	`ssh-copy-id server1_docker`
18. Now you can directly login.	`ssh server1_docker`
19. Go back to terminal.	`logout`
20. Config `tmux` by uploading local files from local to docker server.	`scp .bashrc .vimrc .tmux.conf server server1_docker:`
21. Then go to `tmux`.	`ssh server1_docker`, then `tmux`

Reference: acwing.com