

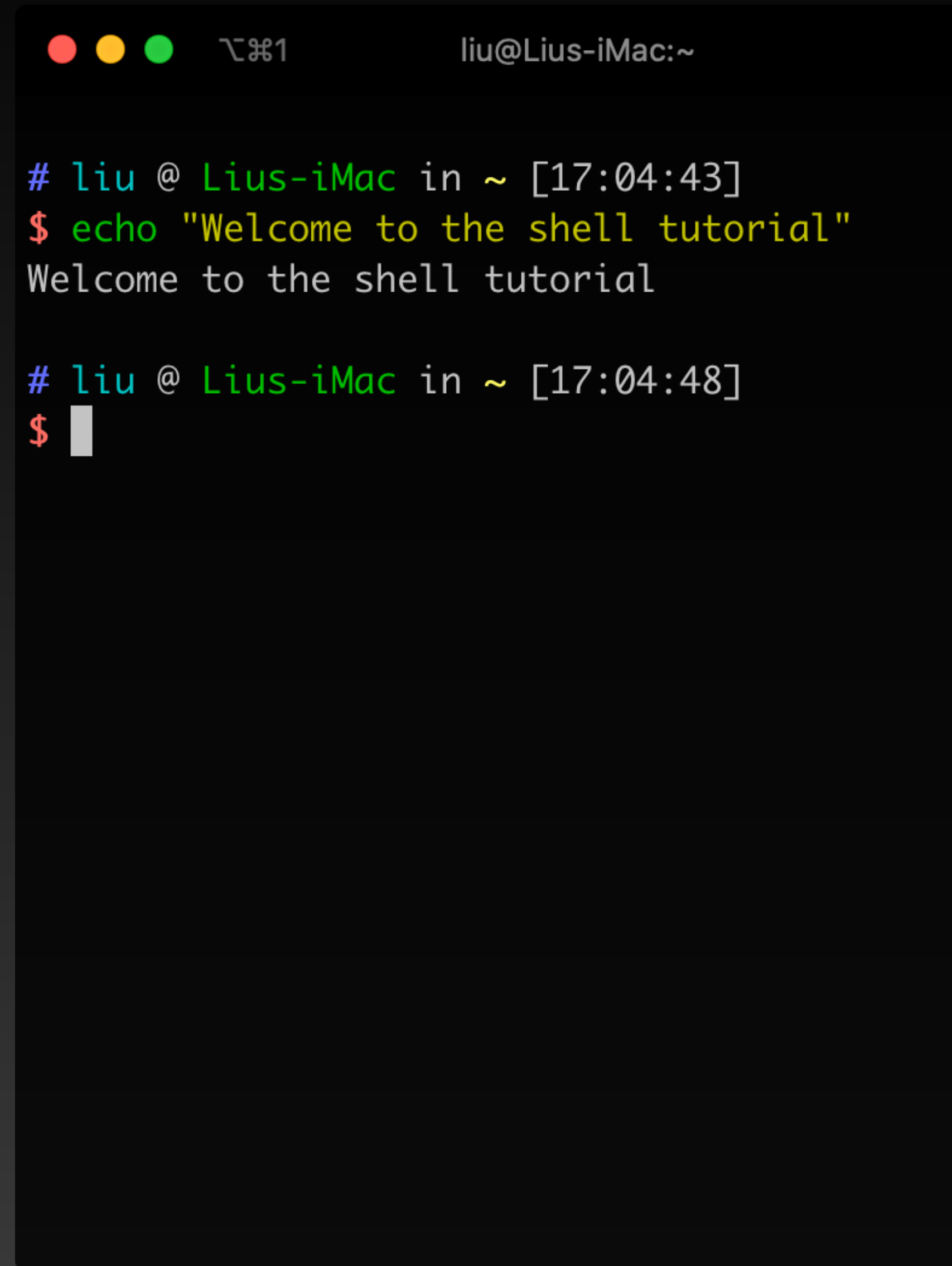
Shell Tutorial

Control/use your computer in a
more efficient (also more geek) way

Nian Liu
nianliu@sjtu.edu.cn

Road Map

- ✓ Brief Intro: all you need to know about starting using a CLI
- ✓ Basic but useful command line tools
- ✓ How to write a bash scripts and what can those scripts do?
- ✓ Real-world examples



```
liu@Lius-iMac:~  
# liu @ Lius-iMac in ~ [17:04:43]  
$ echo "Welcome to the shell tutorial"  
Welcome to the shell tutorial  
  
# liu @ Lius-iMac in ~ [17:04:48]  
$
```

Today's Target

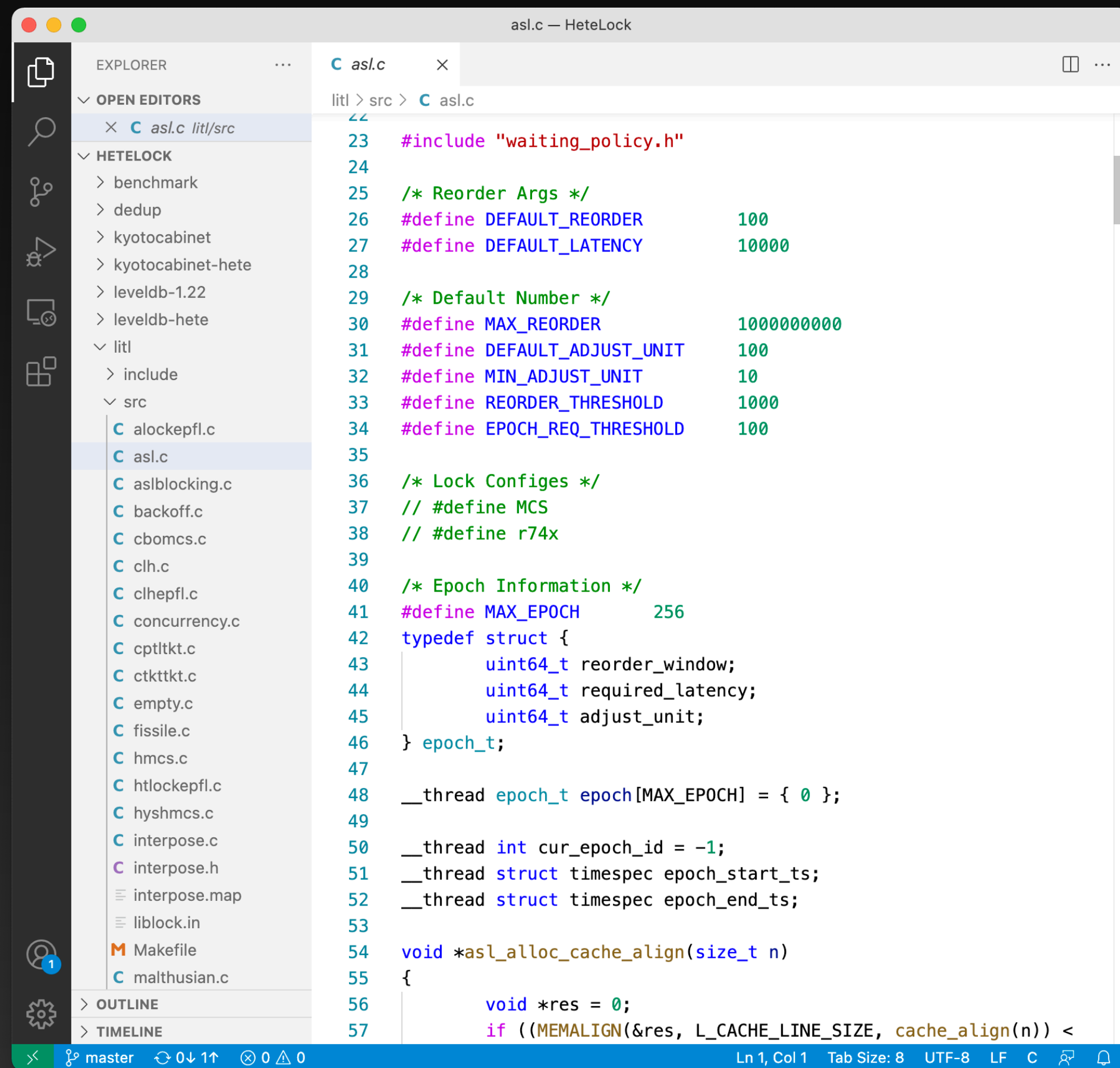
1. Using shell to get the work done (doing experiment, coding, etc.) efficiently
2. Simple Result (data) Processing using bash script
3. Automatic experiment, data collecting and plotting figures

! It is **not** a detailed tutorial, find out more details (e.g., about how to use each tools, more useful commands and techniques) yourself.

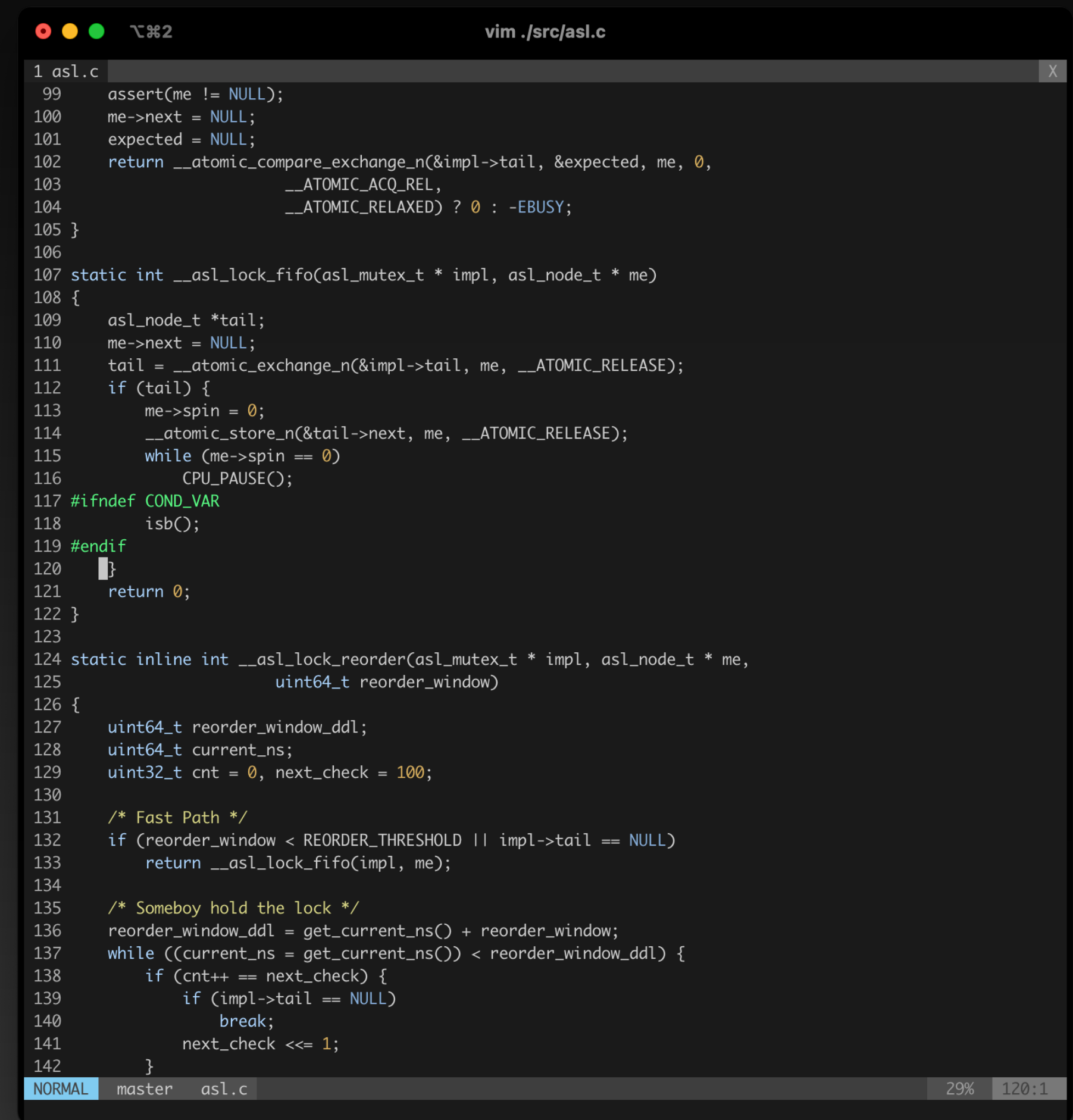
! It is **okey not having this lesson** (like myself). Only a quick start, experience sharing or something like that.

Part 1: CLI? That's cool!

GUI vs CLI: which is better?



```
23 #include "waiting_policy.h"
24
25 /* Reorder Args */
26 #define DEFAULT_REORDER 100
27 #define DEFAULT_LATENCY 10000
28
29 /* Default Number */
30 #define MAX_REORDER 1000000000
31 #define DEFAULT_ADJUST_UNIT 100
32 #define MIN_ADJUST_UNIT 10
33 #define REORDER_THRESHOLD 1000
34 #define EPOCH_REQ_THRESHOLD 100
35
36 /* Lock Configs */
37 // #define MCS
38 // #define r74x
39
40 /* Epoch Information */
41 #define MAX_EPOCH 256
42 typedef struct {
43     uint64_t reorder_window;
44     uint64_t required_latency;
45     uint64_t adjust_unit;
46 } epoch_t;
47
48 __thread epoch_t epoch[MAX_EPOCH] = { 0 };
49
50 __thread int cur_epoch_id = -1;
51 __thread struct timespec epoch_start_ts;
52 __thread struct timespec epoch_end_ts;
53
54 void *asl_alloc_cache_align(size_t n)
55 {
56     void *res = 0;
57     if ((MEMALIGN(&res, L_CACHE_LINE_SIZE, cache_align(n)) <
```



```
1 asl.c
99 assert(me != NULL);
100 me->next = NULL;
101 expected = NULL;
102 return __atomic_compare_exchange_n(&impl->tail, &expected, me, 0,
103     __ATOMIC_ACQ_REL,
104     __ATOMIC_RELAXED) ? 0 : -EBUSY;
105 }
106
107 static int __asl_lock_fifo(asl_mutex_t * impl, asl_node_t * me)
108 {
109     asl_node_t *tail;
110     me->next = NULL;
111     tail = __atomic_exchange_n(&impl->tail, me, __ATOMIC_RELEASE);
112     if (tail) {
113         me->spin = 0;
114         __atomic_store_n(&tail->next, me, __ATOMIC_RELEASE);
115         while (me->spin == 0)
116             CPU_PAUSE();
117 #ifndef COND_VAR
118     isb();
119 #endif
120 }
121 return 0;
122 }
123
124 static inline int __asl_lock_reorder(asl_mutex_t * impl, asl_node_t * me,
125     uint64_t reorder_window)
126 {
127     uint64_t reorder_window_ddl;
128     uint64_t current_ns;
129     uint32_t cnt = 0, next_check = 100;
130
131     /* Fast Path */
132     if (reorder_window < REORDER_THRESHOLD || impl->tail == NULL)
133         return __asl_lock_fifo(impl, me);
134
135     /* Someboy hold the lock */
136     reorder_window_ddl = get_current_ns() + reorder_window;
137     while ((current_ns = get_current_ns()) < reorder_window_ddl) {
138         if (cnt++ == next_check) {
139             if (impl->tail == NULL)
140                 break;
141             next_check <=<= 1;
142         }
143     }
144 }
```

GUI vs CLI: which is better?

GUI

- Graph (e.g., code analysis tools)
- More intuitive user-interface, especially in complex software

CLI

- Data analysis
- Home-brew app
- When connecting to a server

When both are available (e.g., editor), use the one **suit you best!**

Shell: The system user-interface in CLI

Just like the Desktop in GUI world (from user's view)

- Capability: Launch app, execute command, manage foreground/background tasks
- A lot of shell available: zsh, bash, sh, etc.
 - Mostly similar
- Differences: build-in commands, script grammar, extensions
- Chose the one you like
- Useful extensions of oh my zsh: history, autosuggestion, vim-like

```
omz update

plugins/ssh-agent/README.md | 18 +-
plugins/ssh-agent/ssh-agent.plugin.zsh | 5 +-
themes/obraun.zsh-theme | 3 +-
tools/upgrade.sh | 101 ++++++
8 files changed, 411 insertions(+), 300 deletions(-)
Successfully rebased and updated refs/heads/master.
master

Features:
- 9bd0ac9 [mvn] Support using `mvnw` in multi-module projects (#9413)
- dbf5554 [obraun] Display time with leading zeros (#10289)
- a0ac789 [ssh-agent] Allow lazy-loading SSH identities (#6309)
- 19f9b6f [updater] Add support for terminal hyperlinks

Bug fixes:
- 6ac1ff6 [git] Fix directory parse from URL in `gcc` (#10276)
- f82aa81 [lib] Fix `diff --color` argument check for BSD systems (#10269)
- 07cdd7a [lib] Fix status exit code check in `git_prompt_status` (#10275)
- beeda72 [ssh-agent] Fix for bad `zstyle` command argument

You can see the changelog with `omz changelog`

Hooray! Oh My Zsh has been updated!
To keep up with the latest news and updates, follow us on Twitter: https://twitter.com/ohmyzsh
Want to get involved in the community? Join our Discord: https://discord.gg/ohmyzsh
Get your Oh My Zsh swag at: https://shop.planetargon.com/collections/oh-my-zsh

# liu @ MacMini in ~/Projects/HeteLock/litl on git:master o [17:20:44]
$ omz update
```


Basic Setup

- Terminal (emulator): emulate a (texted-based) terminal inside the GUI environment
- SSH to server
 - Running sshd: daemon of SSH server
 - Strong password or use ssh key to login
 - Keep the connection: tmux, screen, etc.
- Keyboard shortcuts
 - ctrl + r (to find history), tab (to autofill)
 - ctrl + c (to kill SIGINT)



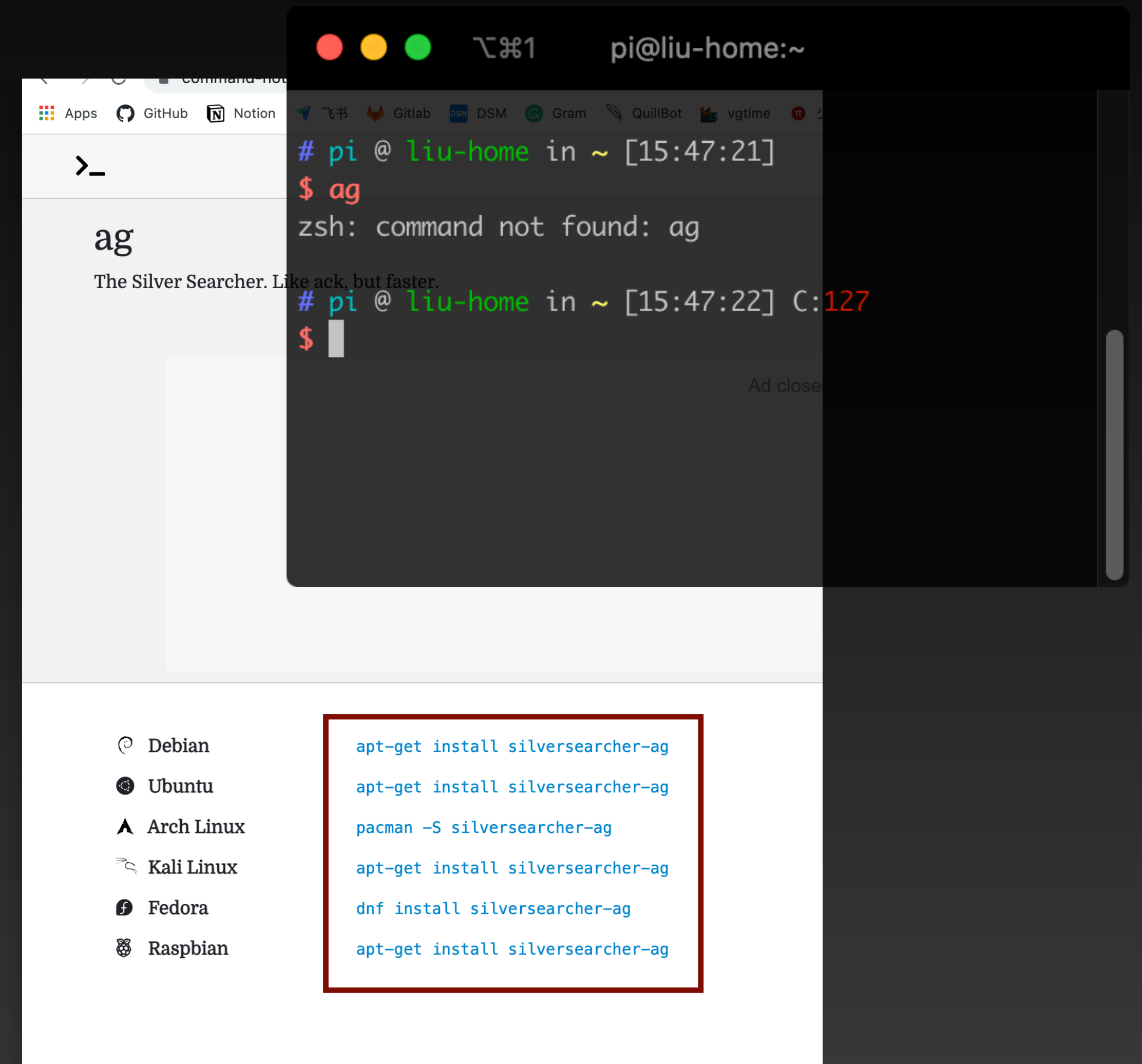
More can be found here !



<https://www.howtogeek.com/howto/ubuntu/keyboard-shortcuts-for-bash-command-shell-for-ubuntu-debian-suse-redhat-linux-etc/>

Install Software in CLI

- Package manager: apt (ubuntu, Debian), brew (macOS), dnf (fedora)
- Search (e.g. apt search)
- <https://command-not-found.com/>
- Build from source (no suitable version, or need to modify their code)
- README/INSTALL doc
- configure and make install



Communication: Pipe & Redirect

- A lot of CLI tools, **communication** is required to do complex jobs
- Pipe: | use the stdout of previous command as the stdin of the next



ls | **grep** "build"

grep: Matches patterns in input text

```
liu@LN-PC:~/projects/linux-build/linux

# liu @ LN-PC in ~/projects/linux-build/linux on git:master o [19:26:37]
$ ls
arch      CREDITS      Documentation  ipc      LICENSES      README      tools
block     crypto       drivers        Kbuild   MAINTAINERS   samples     usr
build     cscope.in.out fs             Kconfig   Makefile      scripts     virt
certs     cscope.out   include        kernel   mm            security
COPYING   cscope.po.out init           lib       net           sound

# liu @ LN-PC in ~/projects/linux-build/linux on git:master o [19:36:39]
$ ls | grep build
build
Kbuild

# liu @ LN-PC in ~/projects/linux-build/linux on git:master o [19:37:02]
$
```

Communication: Pipe & Redirect

- A lot of CLI tools, communication is required to do complex jobs
- Redirect: > & <, stdout to **file** or **file** to stdin (normally)



`ls > ls_out`



`grep build < ls_out`

```
liu@LN-PC:~/projects/linux-build/linux

# liu @ LN-PC in ~/projects/linux-build/linux on git:master o [19:43:05]
$ ls > ls_out

# liu @ LN-PC in ~/projects/linux-build/linux on git:master x [19:43:08]
$ grep build < ls_out
build
Kbuild
```


Communication: Pipe & Redirect

- A lot of CLI tools, communication is required to do complex jobs
- Redirect: > & <, stdout to file or file to stdin (w/o explicitly specified)

```
1 test.c X
1 #include<stdio.h>
2
3 int main(void)
4 {
5     printf("stdout\n");
6     fprintf(stdout, "stdout\n");
7     fprintf(stderr, "stderr\n");
8     return 0;
9 }

# liu @ MacMini in ~/Testspace [19:56:14]
$ ./test > test_out
stderr

# liu @ MacMini in ~/Testspace [19:56:20]
$ ./test > test_out 2>test_err_out

# liu @ MacMini in ~/Testspace [19:56:27]
$ cat test_err_out
stderr

# liu @ MacMini in ~/Testspace [19:56:30]
$
```

NORMAL test.c 77% 7:12

[0] 0:zsh* "MacMini.local" 19:56 20-Oct-21

Part 2: Handy tools make things easier

Basic Tools (Commands)

- File: touch, cp, rm, cat, find, head, tail, less, mkdir, ln
- Simple functions: sort, wc
- How to use?
 - -help, --help
 - man [command]
 - <https://command-not-found.com/>
 - TLDR <https://tldr.sh/>

Find out yourself:

[illegible]

ag

Usage Scenario: Find keyword

```
liu@LN-PC:~/projects/linux-build/linux

# liu @ LN-PC in ~/projects/linux-build/linux on git:master o [19:24:04]
$ ag cpufreq_driver_fast_switch
include/linux/cpufreq.h
589:unsigned int cpufreq_driver_fast_switch(struct cpufreq_policy *policy,

drivers/cpufreq/cpufreq.c
2065: * cpufreq_driver_fast_switch - Carry out a fast CPU frequency switch.
2087:unsigned int cpufreq_driver_fast_switch(struct cpufreq_policy *policy,
2111:EXPORT_SYMBOL_GPL(cpufreq_driver_fast_switch);

kernel/sched/cpufreq_schedutil.c
368:         cpufreq_driver_fast_switch(sg_policy->policy, next_f);
454:         cpufreq_driver_fast_switch(sg_policy->policy, next_f);

# liu @ LN-PC in ~/projects/linux-build/linux on git:master o [19:24:06]
$
```

command-not-found.com/ag

ag

The Silver Searcher. Like ack, but faster.

Maintainer: Hajime Mizuno <mizuno-as@ubuntu.com>

Homepage: https://github.com/ggreer/the_silver_searcher

Section: utils

Ad closed by Google

pkg name

(how to install with pkg mgr)

Examples

Debian

Ubuntu

Arch Linux

Kali Linux

Fedora

Raspbian

apt-get install silversearcher-ag

apt-get install silversearcher-ag

pacman -S silversearcher-ag

apt-get install silversearcher-ag

dnf install silversearcher-ag

apt-get install silversearcher-ag

The Silver Searcher. Like ack, but faster.

Find files containing "foo", and print the line matches in context:
ag foo

Find files containing "foo" in a specific directory:
ag foo path/to/folder

Find files containing "foo", but only list the filenames:
ag -l foo

Find files containing "FOO" case-insensitively, and print only the match, rather than the whole line:
ag -i -o FOO

Find "foo" in files with a name matching "bar":
ag foo -G bar

Find files whose contents match a regular expression:
ag '^ba(r|z)\$'

Find files with a name matching "foo":
ag -g foo

© tl;dr; authors and contributors

<https://command-not-found.com/>

ag

Usage Scenario: Find keyword in code, doc, stdout, etc.

```
liu@LN-PC:~/projects/linux-build/linux

# liu @ LN-PC in ~/projects/linux-build/linux on git:master o [19:24:57]
$ ag "cpufreq_driver_fast.*"
include/linux/cpufreq.h
589:unsigned int cpufreq_driver_fast_switch(struct cpufreq_policy *policy,

drivers/cpufreq/cpufreq.c
2065: * cpufreq_driver_fast_switch - Carry out a fast CPU frequency switch.
2087:unsigned int cpufreq_driver_fast_switch(struct cpufreq_policy *policy,
2111:EXPORT_SYMBOL_GPL(cpufreq_driver_fast_switch);

kernel/sched/cpufreq_schedutil.c
368: cpufreq_driver_fast_switch(sg_policy->policy, next_f);
454: cpufreq_driver_fast_switch(sg_policy->policy, next_f);

# liu @ LN-PC in ~/projects/linux-build/linux on git:master o [19:24:58]
$
```

Also support regex

```
liu@LN-PC:~

# liu @ LN-PC in ~ [19:59:22]
$ ps -A | ag sshd
749 ?      00:00:00 sshd
27385 ?    00:00:00 sshd
27391 ?    00:00:00 sshd
29679 ?    00:00:00 sshd
29685 ?    00:00:00 sshd
31989 ?    00:00:00 sshd
31995 ?    00:00:00 sshd

# liu @ LN-PC in ~ [19:59:28]
$
```

and stdin (pipe, from stdout of other command)

awk

Usage Scenario: Result (data) Processing

- Domain-specific language designed for text processing (c-like)
- Typically used as a data extraction and reporting tool

Normal Use Cases:

- Average, max, min
- Get data in a certain column
- Simple conditional logic

awk

Usage Scenario: Result (data) Processing

Example: Grab Data from a certain column

```
liu@MacMini:~/Testspace
# liu @ MacMini in ~/Testspace [19:14:10] C:127
$ cat tmp
1 a
2 b
3 c
4 d
5 e

# liu @ MacMini in ~/Testspace [19:14:14]
$ cat tmp | awk '{print $2}'
a
b
c
d
e

# liu @ MacMini in ~/Testspace [19:14:29]
$
```

cat tmp | awk '{print \$2}'

\$1	\$2	
1	a	{print \$2}
2	b	execute this code each line
3	c	
4	d	
5	e	

awk

Usage Scenario: Result (data) Processing

Example: Average

```
tmux at

# liu @ MacMini in ~/Testspace [21:24:34]
$ cat tmp
1
2
3
4

# liu @ MacMini in ~/Testspace [21:24:39]
$ cat tmp | awk 'BEGIN{cnt=0} {sum+=$1;cnt+=1} END {print (sum/cnt)}'
3

# liu @ MacMini in ~/Testspace [21:24:39]
$
```

[1] 0:zsh* "MacMini.local" 21:24 20-Oct-21

```
cat tmp | awk '
  BEGIN {cnt=0}
  {sum+=$1;cnt+=1}
  END {print (sum/cnt)}'
```

BEGIN{cnt=0}

\$1



1

2

3

4

5

{sum+=\$1;cnt+=1}

execute this code each line

END {print (sum/5)}

awk

Usage Scenario: Result (data) Processing

Example: Conditional Logic

```
liu@MacMini:~/Testspace
# liu @ MacMini in ~/Testspace [19:16:32]
$ cat tmp
1 a
2 b
3 c
4 d
5 e

# liu @ MacMini in ~/Testspace [19:16:34]
$ cat tmp | awk '{if($1>3) print $2}'
d
e

# liu @ MacMini in ~/Testspace [19:16:44]
$
```

```
cat tmp | awk '{if($1>3) print $2}'
```

A lot more can be done with this simple tool
Use your imagination!

sed

Usage Scenario: Result (data) Processing

- Edit text in a scriptable manner

Example: Get a certain line from a file

```
sed -n '3 p' ./tmp
```

```
tmux at

# liu @ MacMini in ~/Testspace [21:47:16]
$ cat tmp
1 a
2 b
3 c
4 d
5 e

# liu @ MacMini in ~/Testspace [21:47:18]
$ sed -n '3 p' ./tmp
3 c

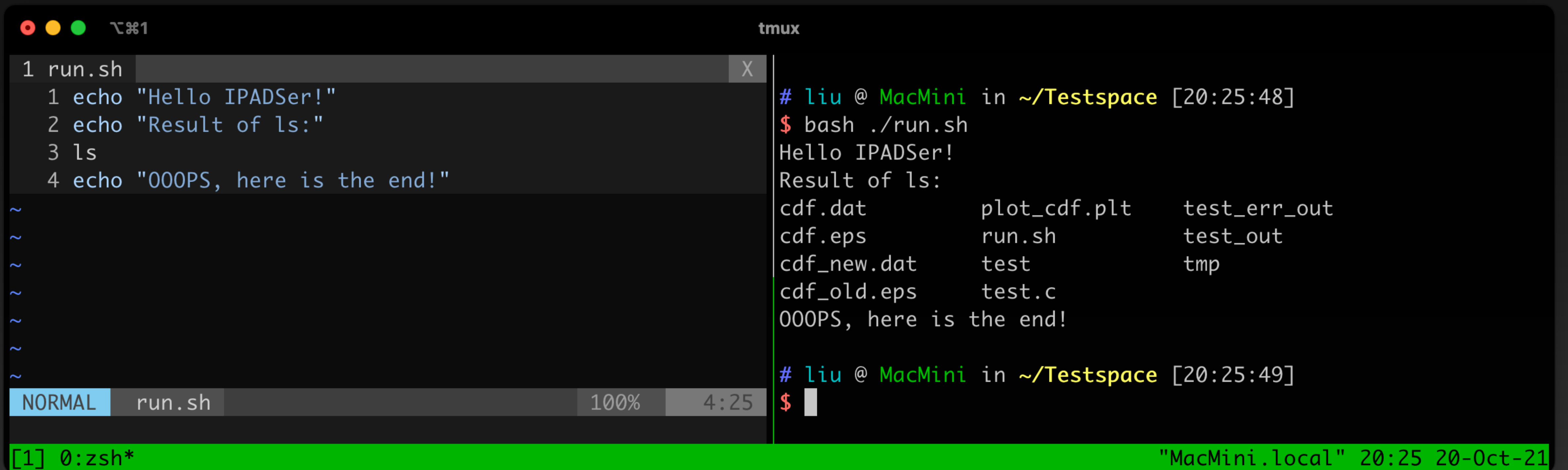
# liu @ MacMini in ~/Testspace [21:47:34]
$ █

[1] 0:zsh* "MacMini.local" 21:47 20-Oct-21
```

Part 3: Lets write some shell scripts!

Shell (Bash) Batch Script

- Basically, **Batch Scripts** are stored in simple text files **containing lines with commands**



The screenshot shows a tmux terminal window with a session named `tmux`. On the left, a vertical pane displays the contents of a file named `run.sh`:

```
1 run.sh
1 echo "Hello IPADSer!"
2 echo "Result of ls:"
3 ls
4 echo "000PS, here is the end!"
```

Below the script content, a status bar indicates the file is `run.sh`, 100% complete, and took 4:25 to execute. The right pane shows the terminal output of running the script:

```
# liu @ MacMini in ~/Testspace [20:25:48]
$ bash ./run.sh
Hello IPADSer!
Result of ls:
cdf.dat          plot_cdf.plt     test_err_out
cdf.eps          run.sh           test_out
cdf_new.dat      test             tmp
cdf_old.eps      test.c
000PS, here is the end!

# liu @ MacMini in ~/Testspace [20:25:49]
$
```

The bottom status bar of the terminal shows the prompt `[1] 0:zsh*` and the system information `"MacMini.local" 20:25 20-Oct-21`.

Shell (Bash) Script

- Basically, **Batch Scripts are** stored in simple text files **containing lines with commands**
- With local variables

The screenshot shows a tmux terminal window with a dark background. The top bar displays window control buttons (red, yellow, green) and the text "tmux". The main pane is split into two sections. The left section shows a script file named "run.sh" with three lines of code: `1 year=2021`, `2 echo "This year is $year!"`, and `3 echo "Last year is ${year-1}!"`. The right section shows the output of running the script, including the prompt `# liu @ MacMini in ~/Testspace [21:04:57]`, the command `$ bash ./run.sh`, and the resulting output lines: `This year is 2021!` and `Last year is 2020!`. Below the script editor, a status bar shows "NORMAL", "run.sh", "100%", and "3:29". At the bottom, a green status bar displays `[1] 0:zsh*` on the left and `"MacMini.local" 21:05 20-Oct-21` on the right.

```

1 run.sh
1 year=2021
2 echo "This year is $year!"
3 echo "Last year is ${year-1}!"

# liu @ MacMini in ~/Testspace [21:04:57]
$ bash ./run.sh
This year is 2021!
Last year is 2020!

# liu @ MacMini in ~/Testspace [21:05:26]
$

NORMAL run.sh 100% 3:29
"./run.sh" 3L, 69B written
[1] 0:zsh* "MacMini.local" 21:05 20-Oct-21

```

Shell (Bash) Script

- Basically, **Batch Scripts are** stored in simple text files **containing lines with commands**
- With local variables
- Passing in as an arguments

```
1 run.sh
1 command=$0
2 prompt=$1
3 year=$2
4
5 echo "$prompt Running $command"
6 echo "$prompt This year is $year"
```

```
# liu @ MacMini in ~/Testspace [20:51:12]
$ bash ./run.sh "[SHELL SCRIPT]" 2021
[SHELL SCRIPT] Running ./run.sh
[SHELL SCRIPT] This year is 2021

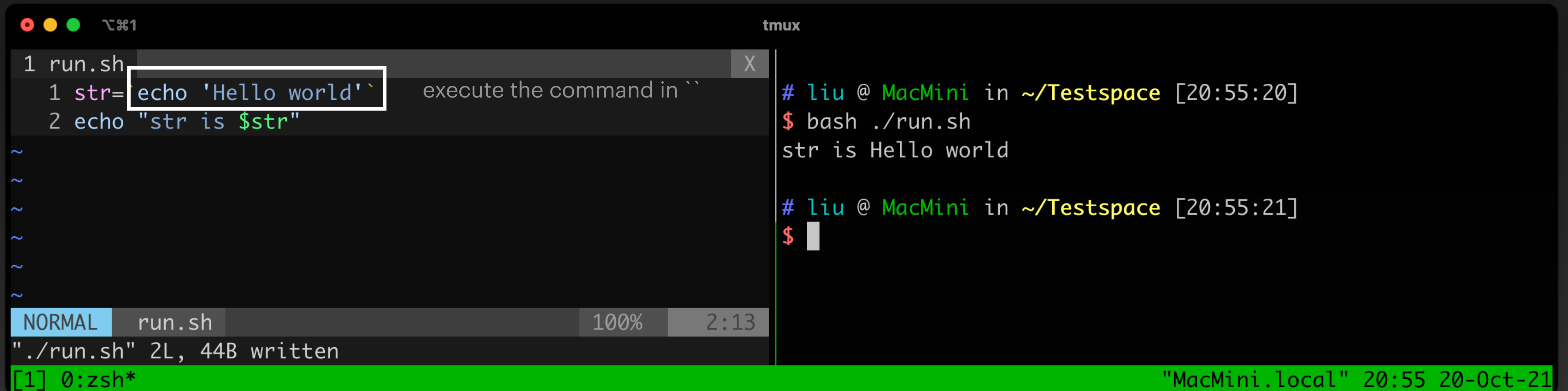
# liu @ MacMini in ~/Testspace [20:51:20]
$
```

The terminal window shows a tmux session. The left pane displays the script 'run.sh' with six lines of code. The right pane shows the execution of the script. The first line of the script sets 'command' to '\$0', 'prompt' to '\$1', and 'year' to '\$2'. The second line is an empty line. The third line echoes the prompt, command, and year. The fourth line echoes the prompt and the year. The terminal output shows the script being executed with arguments '[SHELL SCRIPT]' and '2021'. The output of the script is '[SHELL SCRIPT] Running ./run.sh' and '[SHELL SCRIPT] This year is 2021'. Arrows point from the variable names '\$0', '\$1', and '\$2' to their corresponding values in the terminal output: '\$0' points to '[SHELL SCRIPT]', '\$1' points to '2021', and '\$2' points to '2021'.

[1] 0:zsh* "MacMini.local" 20:51 20-Oct-21

Shell (Bash) Script

- Basically, **Batch Scripts are** stored in simple text files **containing lines with commands**
 - With local variables
 - Passing in as an arguments; or from the results of commands



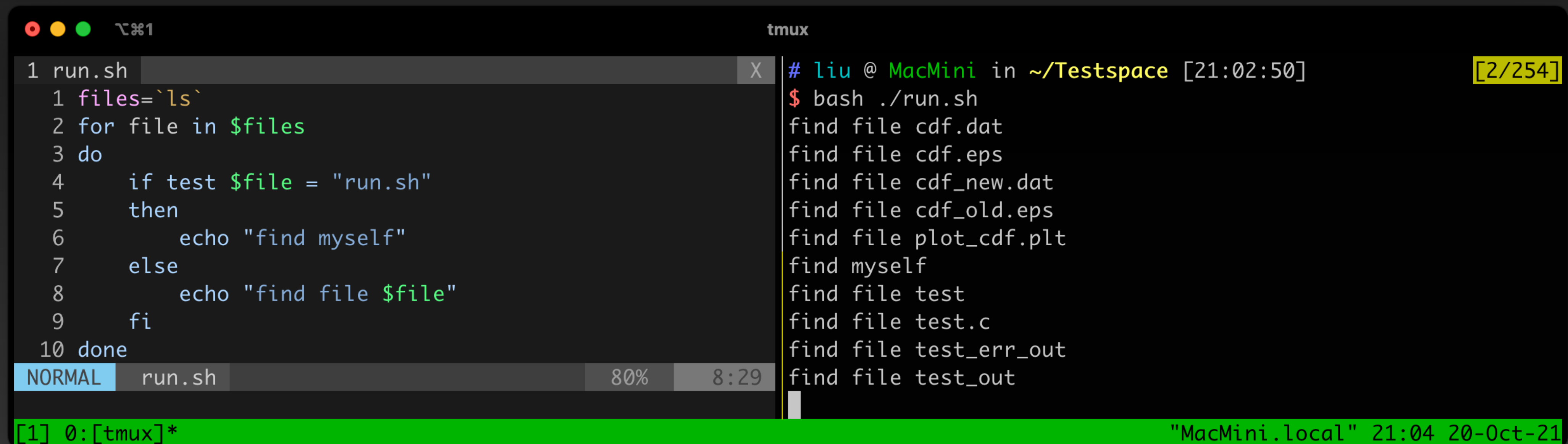
The screenshot shows a tmux terminal window with two panes. The left pane displays a text editor editing a file named `run.sh`. The script content is as follows:

```
1 str=echo 'Hello world'`
2 echo "str is $str"
```

The right pane shows the execution of the script. It starts with a prompt indicating the user is `liu` on `MacMini` in the directory `~/Testspace` at time `[20:55:20]`. The user runs `bash ./run.sh`, and the output is `str is Hello world`. A second prompt at `[20:55:21]` shows the user's shell prompt `$` with a cursor. The bottom status bar of the terminal shows `[1] 0:zsh*` on the left and `"MacMini.local" 20:55 20-Oct-21` on the right.

Shell (Bash) Script

- Basically, **Batch Scripts are** stored in simple text files **containing lines with commands**
- With local variables
- Passing in as an arguments; or from the results of commands
- Support loop and conditions



The screenshot shows a tmux terminal window with two panes. The left pane displays a Bash script named `run.sh` with the following content:

```
1 run.sh
1 files=`ls`
2 for file in $files
3 do
4     if test $file = "run.sh"
5     then
6         echo "find myself"
7     else
8         echo "find file $file"
9     fi
10 done
```

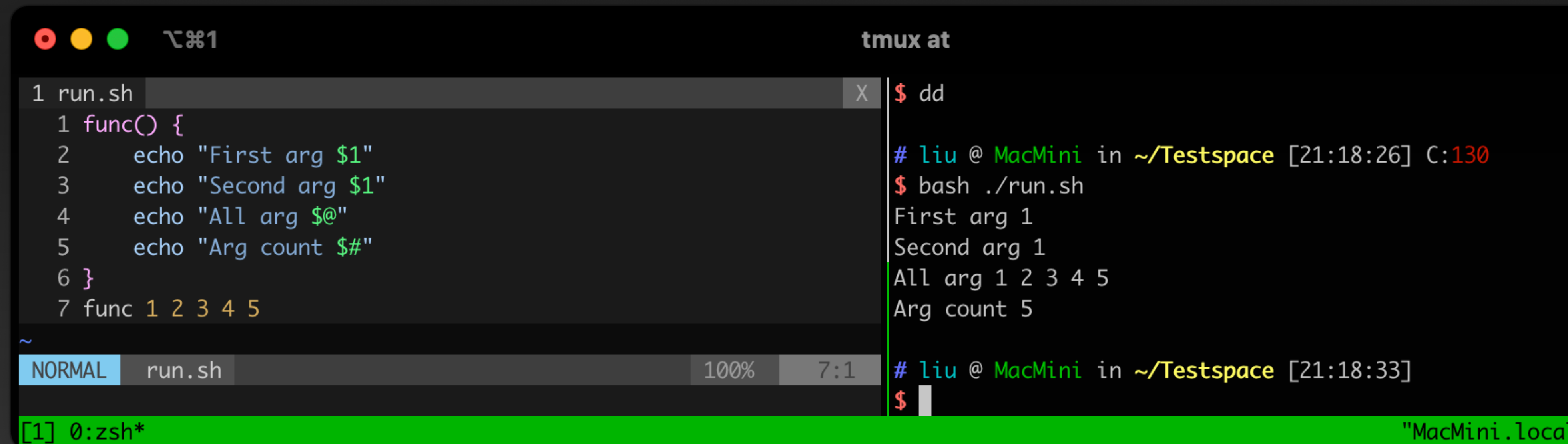
The right pane shows the output of running the script: `$ bash ./run.sh`. The output lists files found by the script:

```
find file cdf.dat
find file cdf.eps
find file cdf_new.dat
find file cdf_old.eps
find file plot_cdf.plt
find myself
find file test
find file test.c
find file test_err_out
find file test_out
```

The terminal window includes a status bar at the bottom with the text `[1] 0:[tmux]*` on the left and `"MacMini.local" 21:04 20-Oct-21` on the right. A yellow box in the top right corner of the right pane contains the text `[2/254]`.

Shell (Bash) Script

- Basically, **Batch Scripts are** stored in simple text files **containing lines with commands**
- With local variables
- Passing in as an arguments; or from the results of commands
- Support loop and conditions
- Functions



The screenshot shows a terminal window with a tmux session. The left pane displays a script named `run.sh` with the following content:

```
1 run.sh
1 func() {
2     echo "First arg $1"
3     echo "Second arg $1"
4     echo "All arg $@"
5     echo "Arg count $#"
```

The right pane shows the execution of the script. It starts with a prompt `$ dd`, followed by a prompt `# liu @ MacMini in ~/Testspace [21:18:26] C:130`. Then, the command `$ bash ./run.sh` is executed, resulting in the following output:

```
First arg 1
Second arg 1
All arg 1 2 3 4 5
Arg count 5
```

Below the output, the prompt `# liu @ MacMini in ~/Testspace [21:18:33]` is shown, followed by a prompt `$`. The bottom status bar of the terminal shows `[1] 0:zsh*` and `"MacMini.local"`.

Shell (Bash) Script

- Basically, **Batch Scripts are** store **commands**
- With local variables
- Passing in as an arguments; or
- Support loop and conditions
- Functions
- Run one after another
- Can call other scripts in a script
- Decoupling

```
tmux at
# liu @ MacMini in ~/Testspace [21:37:33]
$ cat rand.sh
echo $RANDOM
Script 1: generate a random number
# liu @ MacMini in ~/Testspace [21:37:36]
$ cat measure.sh
file=$1
awk 'BEGIN{cnt=0} {sum+=$(1);cnt+=1} END {print (sum/cnt)}' $file
Script 2: avg
# liu @ MacMini in ~/Testspace [21:37:38]
$ cat run.sh
result_file=exp_out
echo "" > $result_file
for i in `seq 0 100`
do
    bash ./rand.sh >> $result_file
done
bash ./measure.sh $result_file
Script 3: use 1 and 2
# liu @ MacMini in ~/Testspace [21:37:42]
$ bash ./run.sh
16712.6
# liu @ MacMini in ~/Testspace [21:37:45]
$
```

[1] 0:bash* "MacMini.local" 21:37 20-Oct-21

**Part 4: Talk is cheap. Show
me some example!**

Running Experiments Multiple Times and Get the Average Result

A screenshot of a tmux terminal window. The window title is "tmux at". The left pane shows a script file named "run_exp.sh" with two lines: "1 echo \"Dummy Experiment Output\"" and "2 echo \"Throughput \$RANDOM ops/s\"". The right pane shows the output of running this script three times. Each run shows the prompt "# liu @ MacMini in ~/Testspace [21:53:51]", followed by the command "\$ bash ./run_exp.sh", and then the output "Dummy Experiment Output" and "Throughput 28935 ops/s", "Throughput 31661 ops/s", and a blank line. The bottom status bar shows "NORMAL run_exp.sh 100% 2:16" and a message "\"./run_exp.sh\" 2L, 63B written". The bottom of the window has a green bar with the text "[1] 0:zsh*" and "MacMini.local 21:53 20-Oct-21".

Dummy Experiment

Example #1

Running Experiments Multiple Times and Get the Average Result

Dummy Experiment Output

Throughput **10592** ops/s

sed get the second line

awk get the number

Q: only use awk?

```
1 run_dummy.sh
2 exp_times=100
3 result_file=tmp_result
4 echo "" > $result_file
5 for i in `seq 1 $exp_times`
6 do
7     bash ./run_exp.sh | sed -n '2 p' | awk '{print $2}' >> $result_file
8 done
9 awk 'BEGIN{cnt=0} {sum+=$1;cnt++;} END{print sum/cnt}' $result_file
```

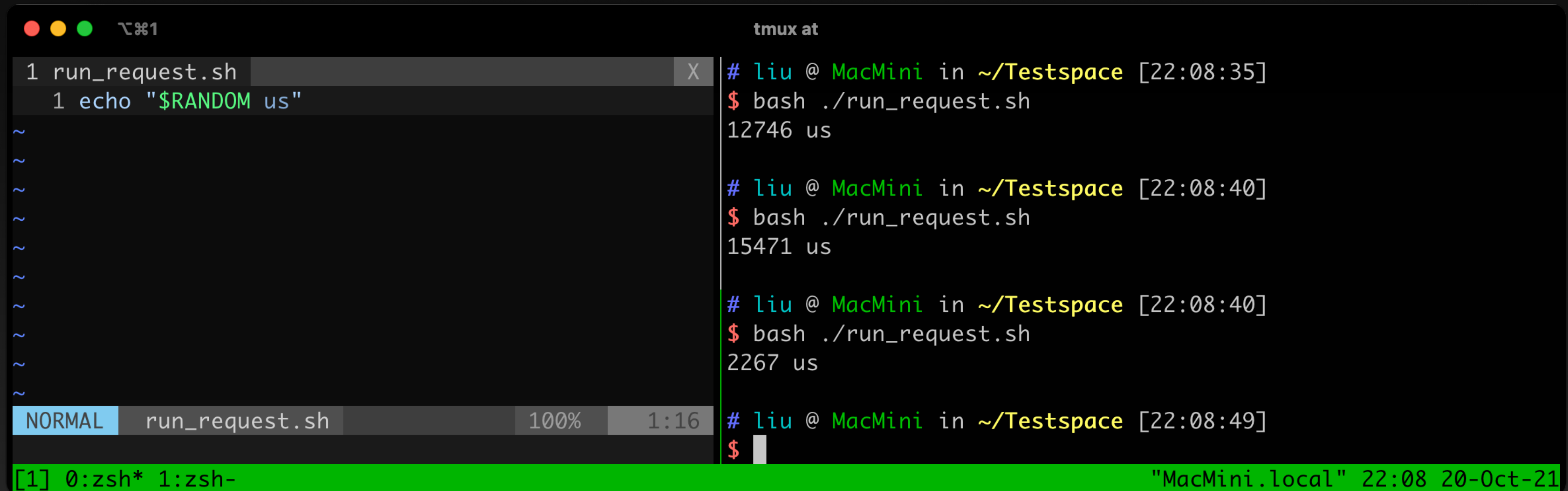
```
# liu @ MacMini in ~/Testspace [21:59:34]
$ bash ./run_dummy.sh
15857.3
```

```
# liu @ MacMini in ~/Testspace [21:59:36]
$
```

Run under different configuration
and use **gnuplot** to plot

```
NORMAL run_dummy.sh 100% 8:67
```

P99, P999 and plot the CDF



Dummy Request Latency

P99, P999 and plot the CDF

A screenshot of a tmux terminal window. The window title is "tmux at". The terminal is split into two panes. The left pane shows a shell script named "run_req_exp.sh" with the following content:

```
1 run_req_exp.sh
1 for i in `seq 1 1000`
2 do
3     bash ./run_request.sh
4 done
```

The right pane shows the output of the script execution:

```
# liu @ MacMini in ~/Testspace [22:25:51]
$ bash ./run_req_exp.sh > req_exp_result

# liu @ MacMini in ~/Testspace [22:25:53]
$ cat ./req_exp_result | head -n 10
1189 us
18027 us
2097 us
18936 us
3006 us
19844 us
3915 us
20753 us
4823 us
21662 us

# liu @ MacMini in ~/Testspace [22:26:04]
$
```

At the bottom of the terminal, there is a status bar. The left part shows "NORMAL", "run_req_exp.sh", "25%", and "1:20". The right part shows the prompt "\$". The bottom of the image shows a green bar with the text "[1] 0:zsh* 1:zsh-" on the left and "MacMini.local" 22:26 20-Oct-21 on the right.

Generate a dummy result

Example #2

P99, P999 and plot the CDF

~ 1

tmux at

```
1 ana.sh X
1 file=$1
2 # Sort the file
3 sort -g $file > $file-sorted
4
5 # Get the result count
6 cnt=`wc -l $file | awk '{print $1}'`
7 echo "Result Count $cnt"
8
9 # p99 p999
10 p99_line=$((cnt*99/100))
11 p999_line=$((cnt*999/1000))
12 p99_lat=`sed -n "$p99_line p" $file-sorted`
13 echo "P99 Latency $p99_lat"
14 p999_lat=`sed -n "$p999_line p" $file-sorted`
15 echo "P999 Latency $p999_lat"
16
17 # cdf
18 awk -v tot_cnt="$cnt" 'BEGIN{line=0} {line++;if (line%2 ==
    0) print (line/tot_cnt" "$1)}' $file-sorted > $file-cdf
```

liu @ MacMini in ~/Testspace [22:25:03]

\$ bash ./ana.sh ./exp_out

Result Count 102

P99 Latency 32085

P999 Latency 32510

liu @ MacMini in ~/Testspace [22:25:05]

\$ cat ./exp_out-cdf | head -n 5

0.0196078 226

0.0392157 1134

0.0588235 2043

0.0784314 2951

0.0980392 3860

liu @ MacMini in ~/Testspace [22:25:07]

\$ cat ./exp_out-cdf | tail -n 5

0.921569 30268

0.941176 30751

0.960784 31602

0.980392 32085

1 32568

Part 5: Whats' next?

The best way to learn it,
is to use it.

Happy shell-ing!

Also read & finish:

<https://missing.csail.mit.edu/2020/course-shell/>

<https://missing.csail.mit.edu/2020/shell-tools/>