SWE202441

SUNG KYUN KWAN UNIVERSITY

Shell Script & Makefile & Git

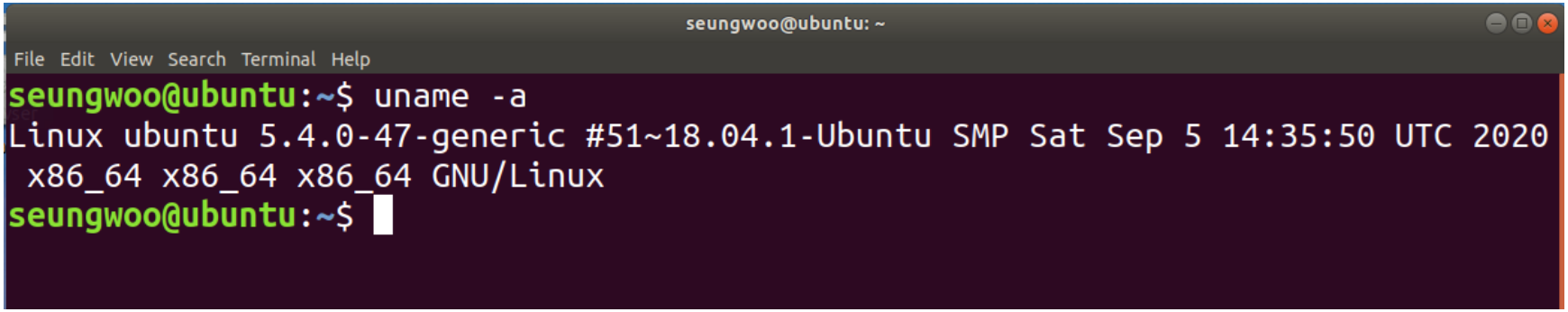
Prof. Joonwon Lee (joonwon@skku.edu)
TA – Jaehyung Park (jaeseanpark@gmail.com)
TA – Luke Albano (lukealbano@arcs.skku.edu)
Sungkyunkwan University

Shell

Interface Between User & OS

What is Shell?

- Interface program between user & OS
  - Similar with Windows command prompt
  - Bash version : $/bin/bash –version
  - Fish, zsh …etc
- Shell takes role of,
  - Control commands
  - Advanced programming language



Redirection & Pipes (1)

- File descriptor
  - Use for process to access file or device
  - Standard file descriptor
    - stdin (0) : standard input (e.g. keyboard)
    - stdout (1) : standard output (e.g. terminal)
    - stderr (2) : standard error output (e.g. error message)
- Redirection
  - Input / output redirection
  - > (overwrite), >> (append)
  - < (stdin)
- Pipe
  - Connect process (propagate stdout to stdin)
  - E.g. cat test.txt | grep “a”

Redirection & Pipes (2)

- ls & ps command

```
$ls –al > output.txt                (redirect result of "ls –al" to output.txt)
$cat output.txt
$ps >> output.txt                (redirect result of "ps" to output.txt)
$cat output.txt
```

- Result

```
total 4
drwxr-xr-x   2 root root   24 Dec  3 01:30 .
dr-xr-x---. 10 root root 4096 Dec  3 01:30 ..
-rw-r--r--   1 root root    0 Dec  3 01:30 output.txt
   PID TTY          TIME CMD
 10679 pts/1    00:00:00 sudo
 10680 pts/1    00:00:00 su
 10681 pts/1    00:00:01 bash
 24822 pts/1    00:00:00 ps
```

Shell Programming

- Two ways of shell programming
  - Line command
  - Execute a shell script

```
$for file in *
>do
>if grep -l ps $file
>then
>less $file
>fi
>done
```

Ex1) Line command

```
$cat test.sh
#!/bin/bash
for file in *
do
        if grep -l ps $file
        then
                cat $file
        fi
done
$bash test.sh
```

Ex2) Execute a shell script

Shell Scripts

- Basically, a shell script is a text file with commands in it
- Shell scripts usually begin with a #! and a shell name
  - For example: #!/bin/bash
- Any command can go in a shell script
  - Commands are executed in order or in the flow determined by control statements
- Different shells have different control structures
- To make script as executable file,
  - $chmod +x [filename]

Grammars of Shell

- Variable
  - String, number, environment, parameter
- Condition
  - Boolean
- Condition control
  - if, elif, for, while, until, case
- List
- Function
- Reversed command

Shell Variable

- Declaration = Initialization of variable
- Case sensitive
- Dereference shell variable by “$”
- Can check value of variable by “echo” command

```
VAR=TEST
echo $VAR
→Hello
VAR=“Hello World”
echo $VAR
→Hello World
read VAR //(get input from stdin)
Best TA ever!
→echo $VAR
Best TA ever!
```

No space!

Shell Predefined Variables

- Some variables are already initialized
- Name of environment variables are <u>upper case</u>
  - Such as $HOME, $PATH, $LD_LIBRARY_PATH
- Different user environment has different value

Predefined Variable	Description
$HOME	Home directory of current user
$PATH	Directory for searching command, divided by “:”
$LD_LIBRARY_PATH	Directory for searching library, divided by “:”
$0	Name of shell script (bash by default)
$#	Number of passed parameter
$$	Process ID of shell script

Shell Parameter Variable

- If shell script run with parameter variables,
  - We can access to parameter as $1, $2, … in script

```
$ cat test.sh
#!/bin/bash
echo $1
echo $2
echo $3
$./test.sh Shell Script Test
Shell
Script
Test
```

Shell Conditional Statement (1)

- If
  - Check condition and executes command block
  - Also we can use else & elif (else if)

```
if condition
then
        statement
        …
elif condition
then
        statement
        …
else
        statement
        …
fi
```

```
#!/bin/bash
read num
if [ $num -lt 5 ]
then
        echo “Lower than 5”
elif [ $num -gt 8 ]
then
        echo “Greater than
8”
else
        echo “5~8”
fi
```

Shell Conditional Statement (2)

- File test
  - -e: True if file exists
  - -d: True if file exists and is a directory
  - Usage: if [-e file.txt]
- String test
  - =, !=, <, >
  - Usage: if [<STRING1> != <STRING2>]
- Arithmetic test
  - -eq (equal), -ne (not equal), -le (less or equal than), -ge (greater or equal than), -lt (less than), -gt (greater than)
  - Usage: if [<INTEGER1> -eq <INTEGER2>]

For-loop

- For
  - Iterate for range of values
  - Range of values can be set of strings

```
for variable in values
do
        statements
done
```

```
#!/bin/bash
for x in a b c d e
do
        echo $x
done
```

```
a
b
c
d
e
```

While-loop

- For-loop is hard to use for fixed number of iteration
- While-loop

```
#!/bin/bash
for x in 1 2 3 4 5 6 7 8 9 10 11 12
do
        echo $x
done
```

while *condition*
do
        statements
done

```
#!/bin/bash
x=1
while [ $x -le 12 ]
do
        echo $x
        ((x++))    same as x=$(($x+1))
done
```

Until-loop

- Until-loop iterates statements until condition becomes true

```
#!/bin/bash
x=1
while [ $x -le 12 ]
do
        echo $x
        ((x++))
done
```

```
until condition
do
        statements
done
```

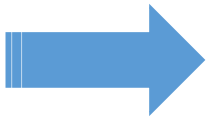
```
#!/bin/bash
x=1
until [ $x -gt 12 ]
do
        echo $x
        ((x++))
done
```

Case Statement

- Execute statements according to the pattern of variable

```
case variable in
        pattern [|pattern] …) statements;;
        pattern [|pattern] …) statements;;
        …
esac
```

```
#!/bin/bash
read input
case $input in
        yes|y|Yes|YES) echo “YES!”;;
        [nN]*) echo “NO!”;;
        *) echo “bad input”;;
esac
```



```
yes
YES!
```

```
y
YES!
```

```
N
NO!
```

```
chicken
bad input
```

Makefile

Simple Way for Compiling

Why Makefile?

- Simplify compiling source codes
- Describe the relationships among files
- Provide commands for updating each file
- Recompile each changed file

```
gcc -o test main.c test.c hello.c
```

vs

```
make
```

Example

Target 'hello' depends on 'main.o' and 'hello.o'

```
# Makefile

hello: main.o hello.o
        gcc -o hello main.o hello.o
main.o: main.c
        gcc -c main.c
hello.o: hello.c
        gcc -c hello.c
clean:
        rm *.o hello
```

Target 'clean' is not a file, but it is the name of an action

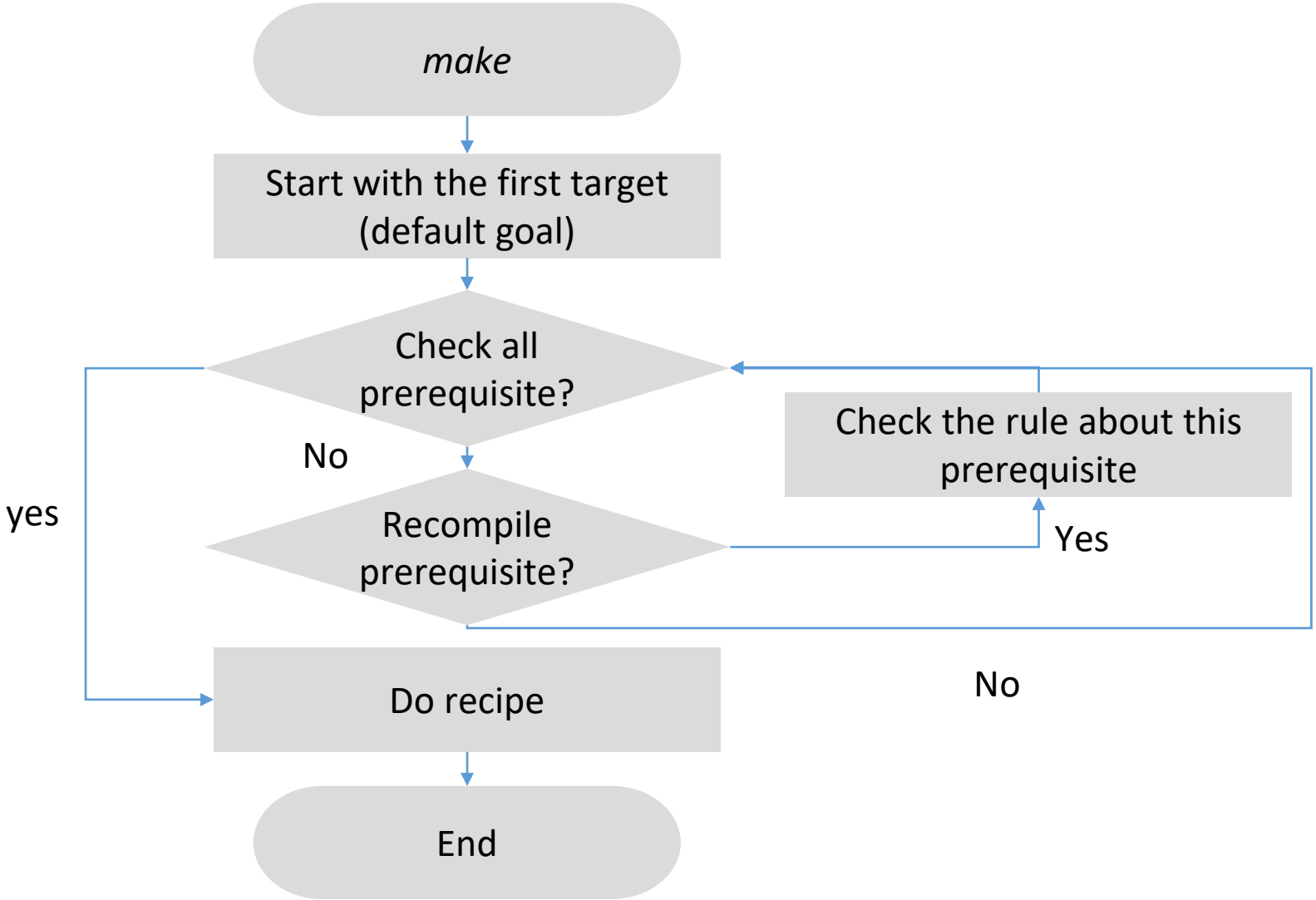
```
$ make
$ ls
$ hello main.c hello.c main.o hello.o
```

Rule

```
target ... : prerequisites ...
    recipe
    ...
    ...
```

- A *rule* explains how and when to remake certain files, or to carry out an action
- A *target* is the name of a file that is generated by a program, or the name of an action to carry out (ex. clean, install)
- A *prerequisite* is a file that is used as input to create the target
- A *recipe* is an action that 'make' carries out

Process of *make*



How *make* process

- If ‘main.c’ is modified and enter make command

```
# Makefile
hello: main.o hello.o      (1) Find default goal
        gcc  -o hello main.o hello.o      (5) Final recipe
main.o: main.c      (2) Check
        gcc  -c main.c      (3) recipe
hello.o: hello.c      (4) Check
        gcc  -c hello.c
clean:
        rm *.o hello
```

Variable

- Be defined once and substituted in multiple places
- Substitute the variable's value by writing $(variable)

```
# Makefile
TARGET=hello
CXX=gcc
OBJECTS=main.o hello.o
$(TARGET): $(OBJECTS)
	$(CXX) -o $(TARGET) $(OBJECTS)
main.o: main.c
	$(CXX) -c main.c
hello.o: hello.c
	$(CXX) -c hello.c
clean:
	rm $(OBJECTS) $(TARGET)
```

Automatic Variables

- $@ : the file name of the target of the rule
- $^ : the names of all the prerequisite
- $? : the names of all the prerequisites that are newer than the target
- $< : the name of first prerequisite

```
$(TARGET): $(OBJECTS)
        $(CXX) -o $@ $^
main.o: main.c
        $(CXX) -c $^
hello.o: hello.c
        $(CXX) -c $^
```

Special Built-in Targets

- .PHONY
  - This target is not really the name of a file
  - Two reasons to use a phony target
    - Avoid conflict with a file of a same name
    - Improve performance
- Others
  - .SUFFIXES, .DEFAULT, .POSIX, etc.

```
.PHONY: clean
clean:
        rm *.o hello
```

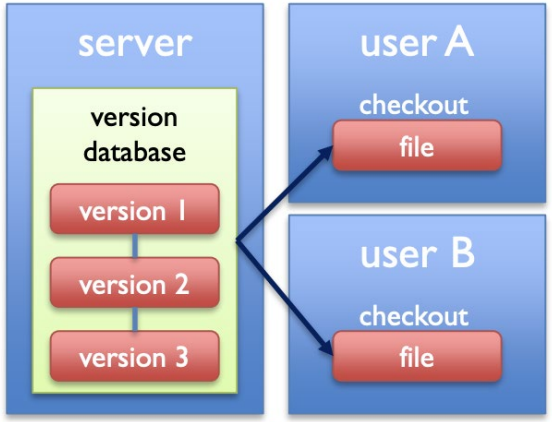
Git

Distributed Version Control System

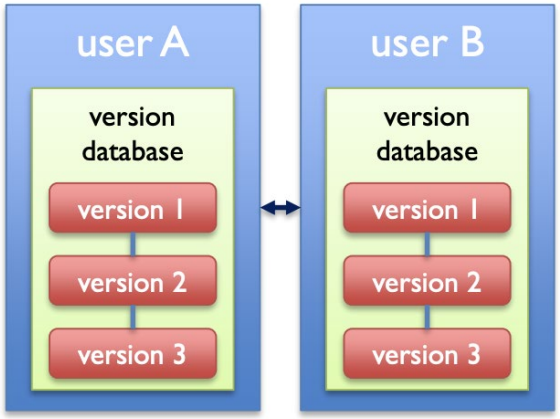
Version Control System

- Manage changes of documents, computer programs, and other collections of information

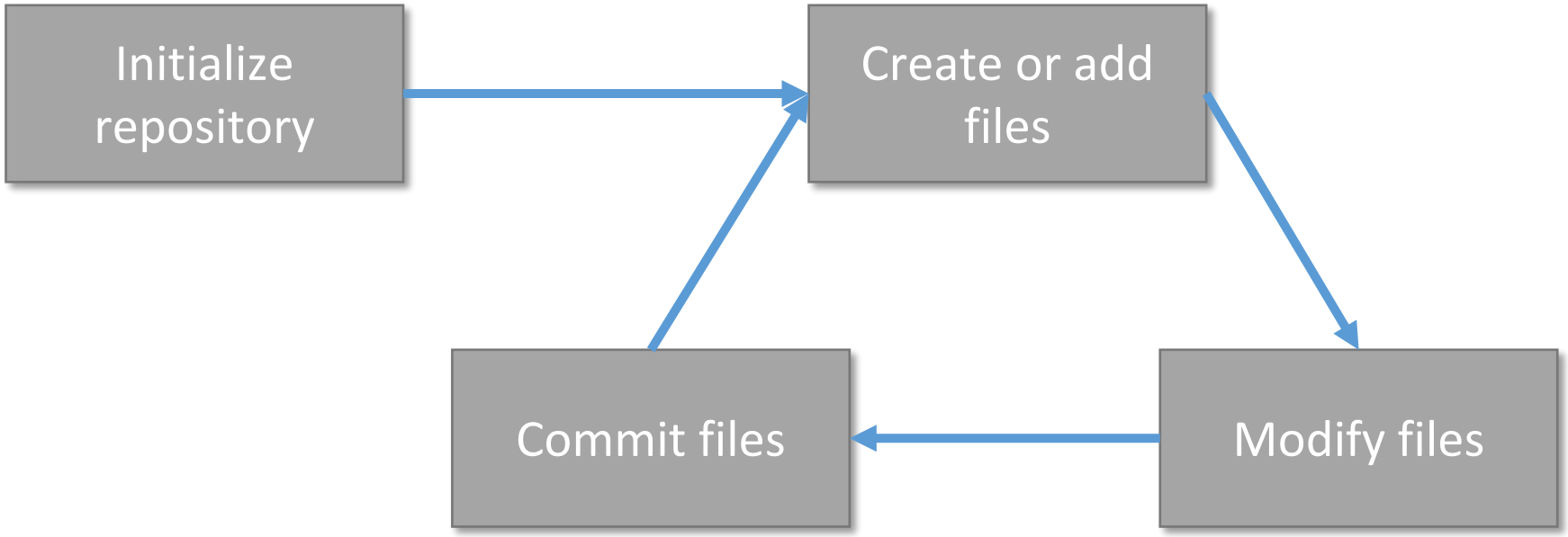
Centralized revision control system



Distributed revision control system



Work Flow on Local Repository



Install & Setup

- Install (ALREADY INSTALLED IN THE INUIYEJI!)
  - Linux : sudo apt install git
- Windows, Mac
  - download at http://git-scm.com/
- User setup
  - git config --global user.name “name”
  - git config --global user.email “e-mail”

Git Basic Command

- git init
  - Create an empty Git repository or reinitialize an existing one
- git add “filename”
  - Add file contents to the index
- git rm “filename”
  - Remove files from the working tree and from the index
- git commit
  - Record changes to the repository
  - options
    - *-a* : Tell the command to automatically stage files that have been modified
    - *-m “msg”* : Use the given “msg” as the commit message
- git status
  - Show the working directory status

Example

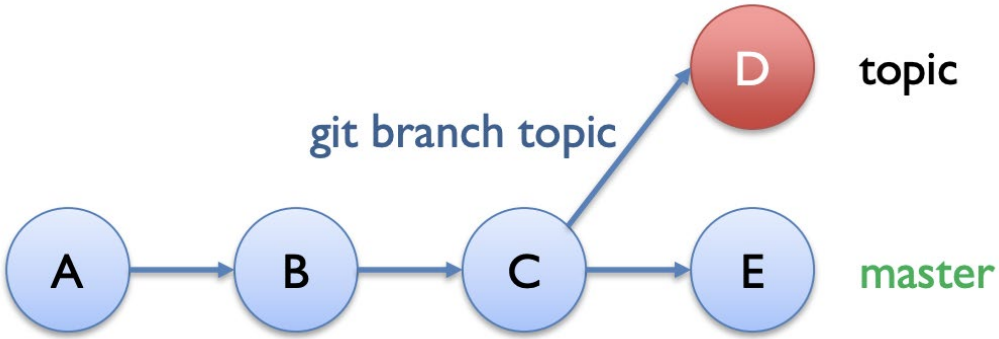
```
$mkdir git_tutorial && cd git_tutorial
~/git_tutorial git init
~/git_tutorial vi hello.c
~/git_tutorial git add hello.c
~/git_tutorial git status
~/git_tutorial git commit
~/git_tutorial vi hello.c
~/git_tutorial git commit –m "Modify hello.c file"
```

```
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   hello.c
```

```
Create hello.c file
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   new file:   hello.c
#
```

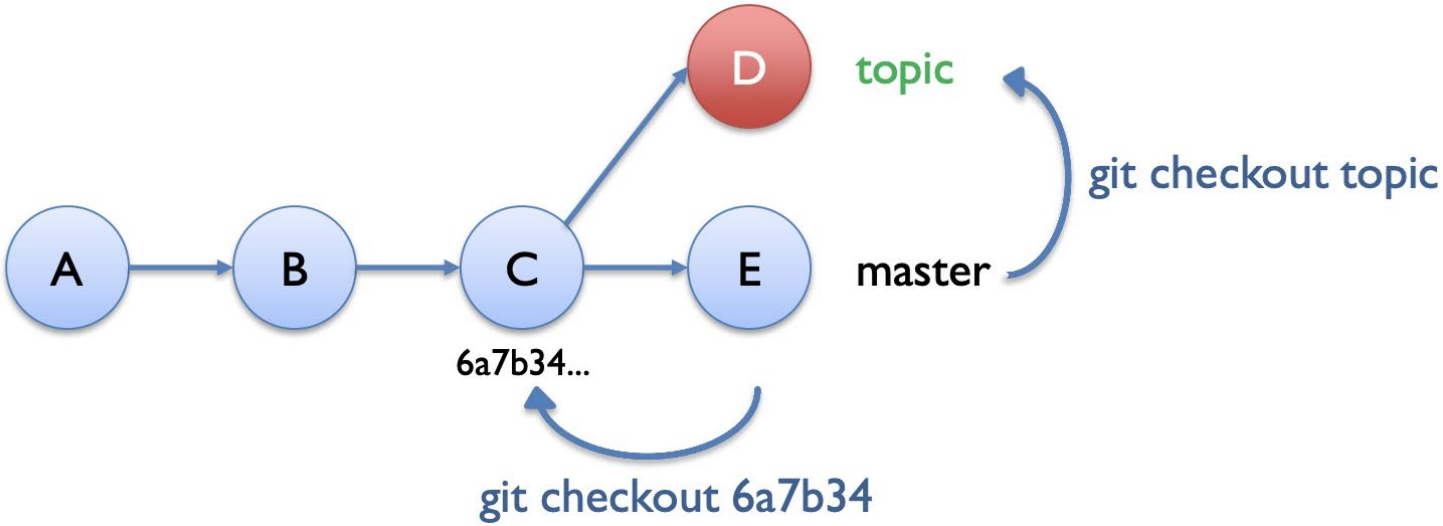
Git Branch Command (1)

- git branch
  - List, create, or delete branches
  - options
    - [-d] “branchname” : delete the branch named “branchname”



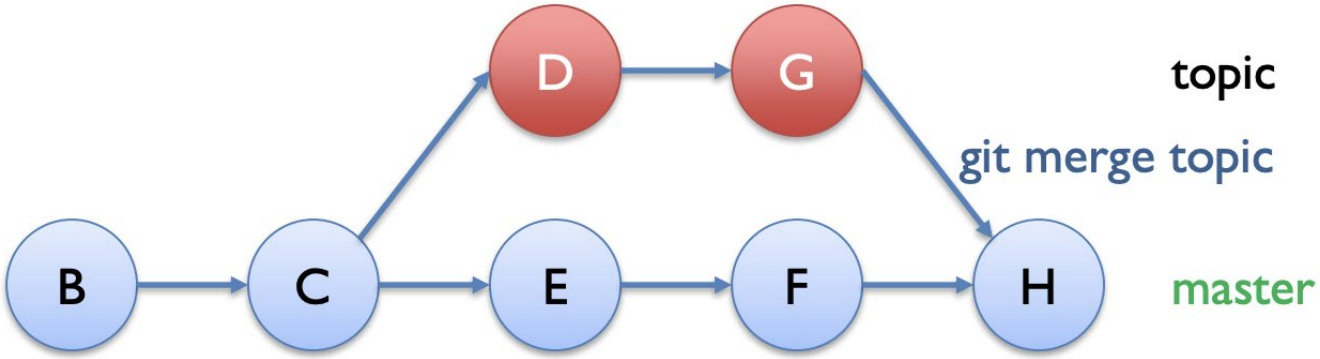
Git Branch Command (2)

- git checkout
  - Checkout a branch or paths to the working tree
  - options
    - “branchname” : switch to “branchname”
    - -b “newbranch” : create “newbranch” and switch



Git Branch Command (3)

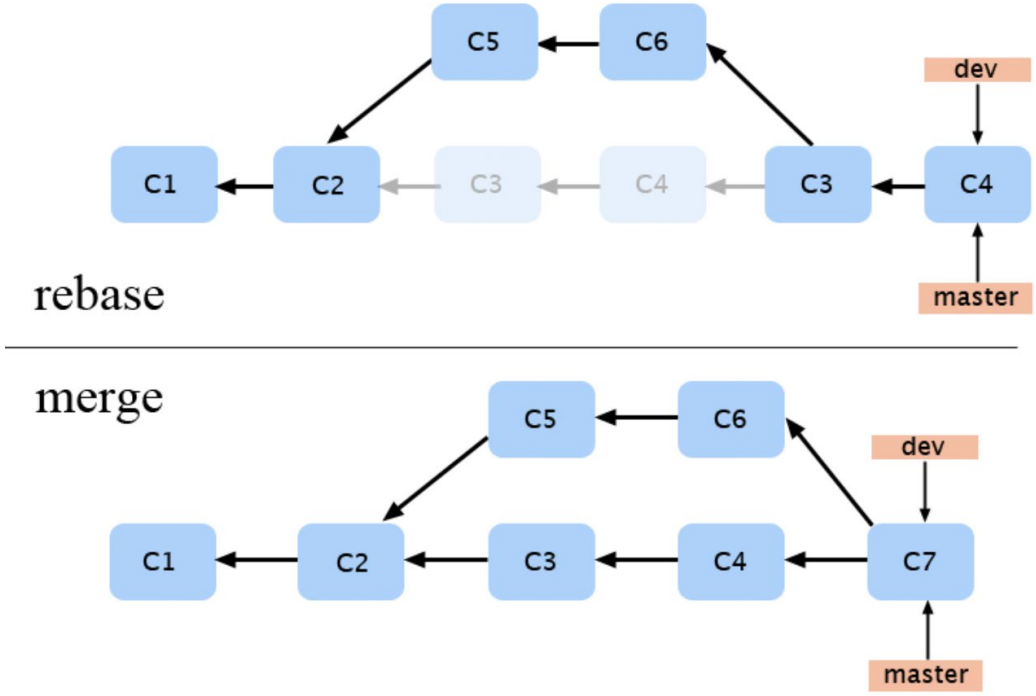
- git merge
  - Join two or more development histories together
  - options
    - “branchname” : Apply the changes of “branchname” on top of current branch



Git Branch Command (4)

- git rebase
  - Forward-port local commits to the updated upstream head

- git merge vs git rebase



Conflict

master

```
#include <stdio.h>

int main(void) {
        printf("Hello!\n");
        printf("Master!\n");
}
```

topic

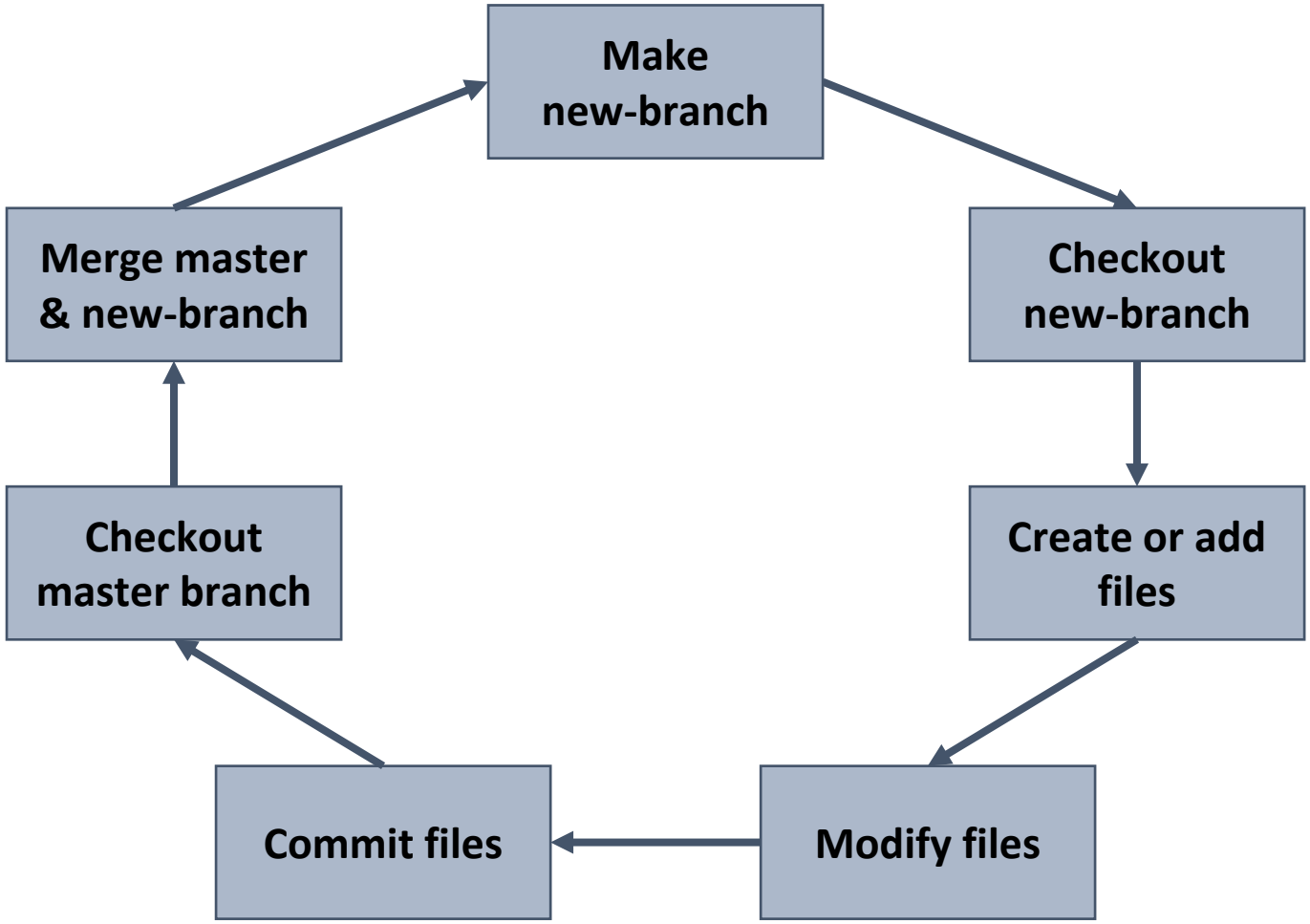
```
#include <stdio.h>

int main(void) {
        printf("Hello!\n");
        printf("Topic!\n");
}
```

git merge topic

```
#include <stdio.h>

int main(void) {
        printf("Hello!\n");
<<<<<<< HEAD
        printf("Master!\n");
=======
        printf("Topic!\n");
>>>>>>> topic
}
```

Fix conflict and commit

Work Flow on Local Repository

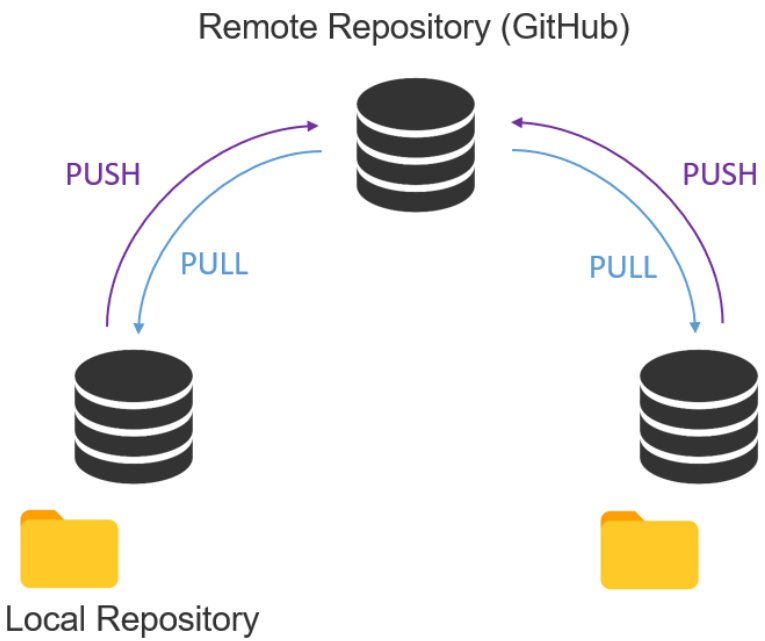


Git Log Command

- git log
  - Show commit logs
  - options
    - -p : Show all changes at each commit
    - --stat : Show statistics about modified files at each commit
    - --name-only : Show only modified file name at each commit
    - --relative-date : Show commit log with relative date
    - --graph : Draw a text-based graphical representation of the commit history

GitHub

- Remote repository (place of co-work)
- Sign up for GitHub
  - https://github.com
- Public repository for free user



GitHub Basic Command (1)

- git remote
  - Link local repository and remote repository
  - options
    - -v : Check the connection with local and remote repository
    - add “name” “url” : Add a remote named “name” for the repository at “url”
- git diff
  - Show changes between local and remote
- git push
  - Push local repository contests to remote repository
  - options
    - “repository” : destination (name or url)

.gitignore

- .gitignore
  - Ignore auxiliary files such as logs, input/out data, etc
  - Generate automatically at https://www.gitignore.io/

```
# no .a files
*.a
# but do track lib.a, even ignoring .a files above
!lib.a
# ignore all files in the build/ directory
build/
```

```
git add .gitignore
git commit –m “added ‘.gitignore’ file”
```

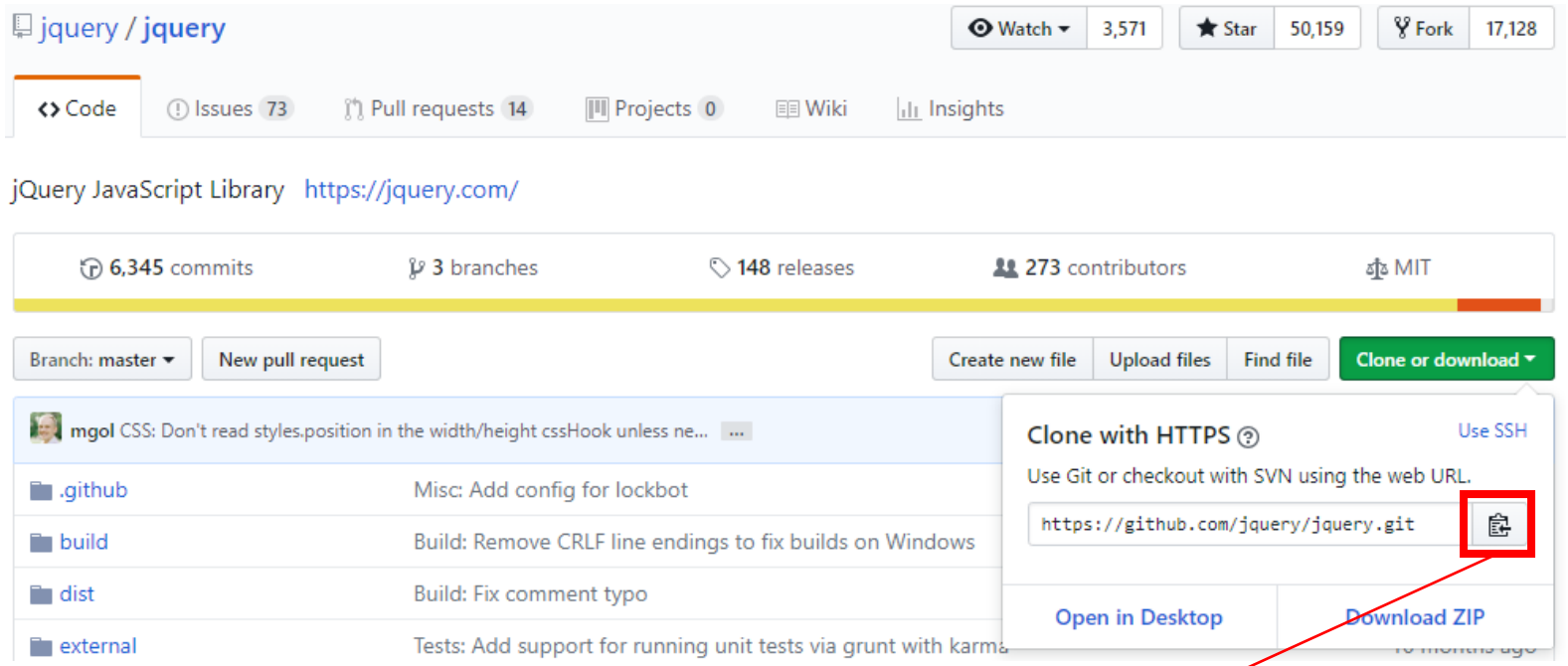
.gitignore

- .gitignore
  - Ignore auxiliary files such as logs, input/out data, etc
  - Generate automatically at https://www.gitignore.io/


```
git add .gitignore
git commit –m “added ‘.gitignore’ file”
```

GitHub Basic Command (2)

- git clone
  - Copy remote repository to local repository




```
~/git_tutorial git clone https://github.com/jquery/jquery.git
```

GitHub Basic Command (3)

- git pull
  - Fetch and integrate contents from remote repository
  - options
    - “repository” : name or url of remote repository

Local repository	Remote repository (GitHub)
	Modified by other user
Commit	
Push (fail)	
Pull	
Push (success)	
	Updated by me

GitHub Basic Command (4)

- git fetch
  - Fetch contents from remote repository
  - options
    - “repository” : name or url of remote repository

Local repository	Remote repository (GitHub)
	Modified by other user
Commit	
Push (fail)	
fetch	
merge	
Push (success)	
	Updated by me

Exercise 1 - Makefile

Write your own *makefile* to compile *main.c*, *plus.c,* and *minus.c* created last week

Filename: [Makefile]

- Use one or more **variable**
- Use one or more **automatic variable**
- Make *clean* rule which removes **object files** and **executable file**
- Check slides 24~25 for hints

Exercise 2 – Shell Script

Hint 1 : *echo* and redirect (>, >>)

Hint 2: Use *man* command when you want to know some commands

Make shell script for scoring your “Plus&Minus”

File name: [score.sh]

The following process should be contained in “score.sh”:

1) Compile your program with *makefile* created in Exercise 1
2) Make answer sheet as answer.txt (use 22 11 as input values)
3) Run your program and save output as output.txt
4) Compare answer.txt and output.txt with ***diff*** command as result.txt
5) Print the contents of “result.txt” with ***cat*** command
6) Clean object files and executable file

Exercise 2 – Shell Script

```
2014311240@swji:/home/2014311240/p3$ ls
main.c  Makefile  minus.c  plus.c  score.sh
2014311240@swji:/home/2014311240/p3$ ./score.sh
gcc -c main.c
gcc -c plus.c
gcc -c minus.c
gcc -o p2.out main.o plus.o minus.o
rm main.o plus.o minus.o p2.out
2014311240@swji:/home/2014311240/p3$ ls
answer.txt  main.c  Makefile  minus.c  output.txt  plus.c  result.txt  score.sh
```

"ls" Before running "score.sh"

"ls" After running "score.sh"

output.txt

```
33 11
```

answer.txt

```
33 11
```

If your program is correct, nothing is written and printed in result.txt

result.txt

Submission

- Submit your Makefile & score.sh as p3
  - InUiYeJi cluster
  - Remove the text files (rm *.txt)
  - Submit the folder into p3
  - ~swe2024-41_23s/bin/submit p3 p3

```
2014311240@swji:/home/2014311240$ ~swe2024-41_23s/bin/submit p3 p3
user name          :2014311240
Submitted Files for p3:
File Name                                File Size       Time
--------------------------------------------------------------------------
p3-2014311240-Mar.15.15.50.911376889     425             Wed Mar 15 15:50:32 2023
```

./p3

- Makefile
- score.sh
- main.c
- plus.c
- minus.c

Skeleton code of score.sh

- Quiz 1 about 3), due 23:59
- copy the skeleton code to your directory

$ cp ~swe2024-41_23s/2023s/p3_skeleton.sh ./

```
#!/bin/bash

# 1) compile your program with makefile created in Exercise 1
# your code here

# 2) make answer sheet as answer.txt
# your code here

# 3) run your program and save output as output.txt
echo "22 11" | ./p2.out > output.txt

# 4) compare answer.txt and output.txt with diff command as result.txt
# your code here

# 5) print the contents of "result.txt"
# your code here

# 6) clean object files and executable file
# your code here
```

Minor adjustments in main.c

- Adding 2 line of codes
- extern int plus(int a, int b);
- extern int minus(int a, int b);
- will get rid of the gcc warnings

```c
#include <stdio.h>

extern int plus(int a, int b);
extern int minus(int a, int b);

int main()
{
    int a, b;
    scanf("%d %d", &a, &b);
    printf("%d %d\n", plus(a, b), minus(a, b));
    return 0;
```