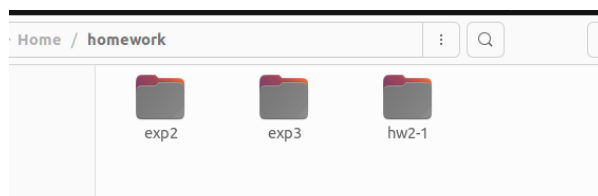
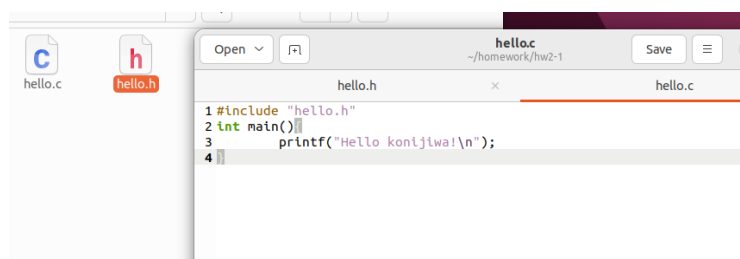
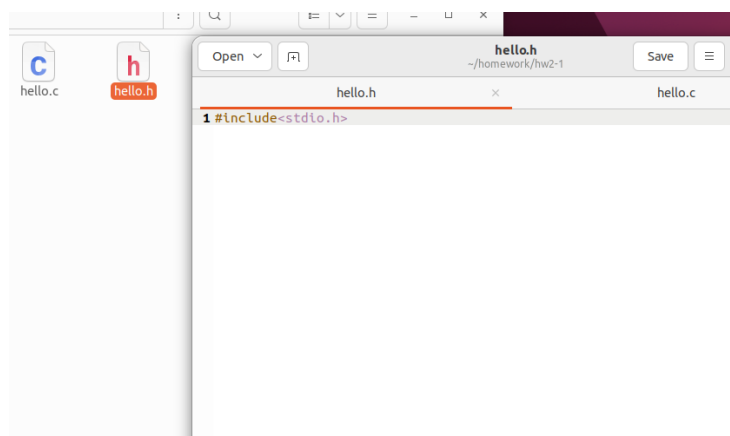
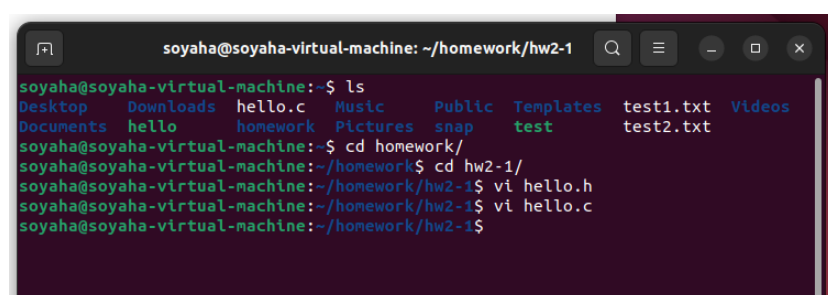


作业 2-1:makefile 工程管理器的使用

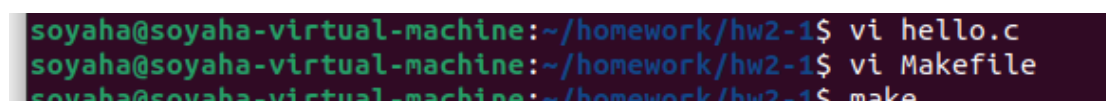
先提前创建好作业文件夹

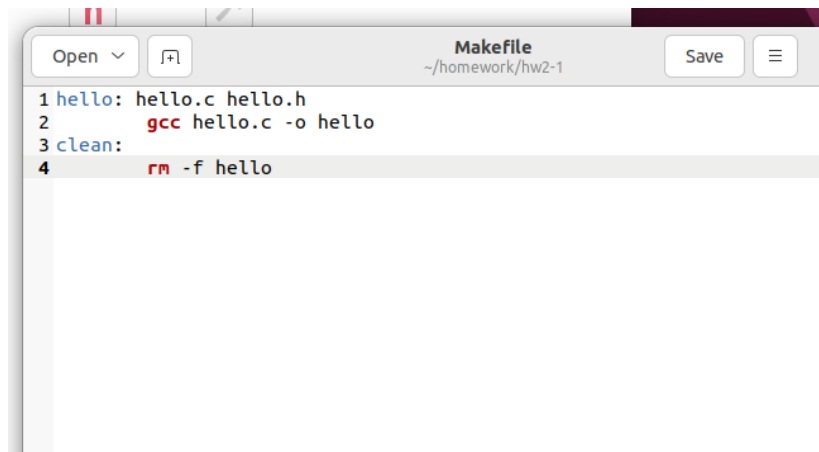


1.然后使用 vi 编辑出 hello.c 和 hello.h 文件

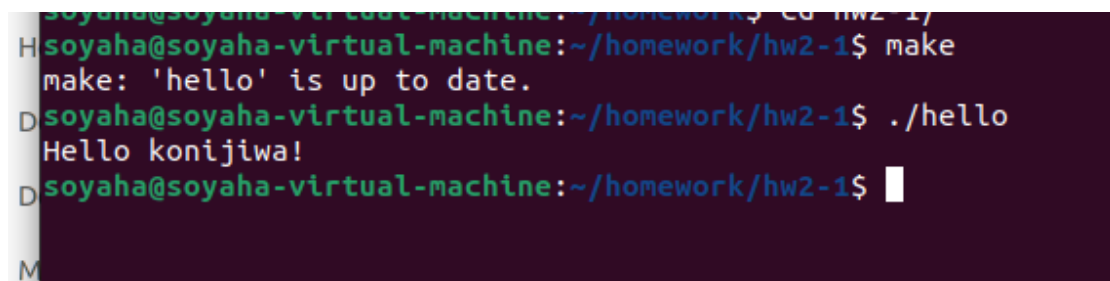


2.再编写无变量、单目标的 Makefile 文件并验证





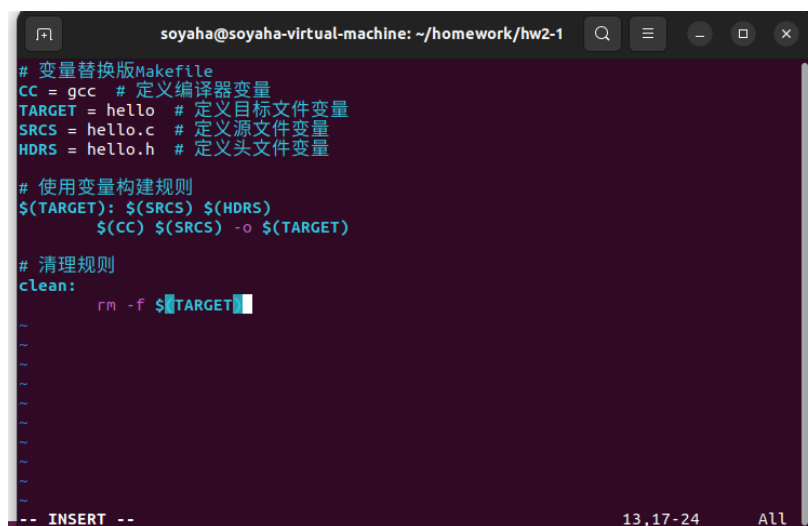
```
1 hello: hello.c hello.h
2     gcc hello.c -o hello
3 clean:
4     rm -f hello
```



```
soyaha@soyaha-virtual-machine: ~/homework/hw2-1
H soyaha@soyaha-virtual-machine:~/homework/hw2-1$ make
make: 'hello' is up to date.
D soyaha@soyaha-virtual-machine:~/homework/hw2-1$ ./hello
Hello konijiwa!
D soyaha@soyaha-virtual-machine:~/homework/hw2-1$
```

可以发现成功编译运行了 hello 程序

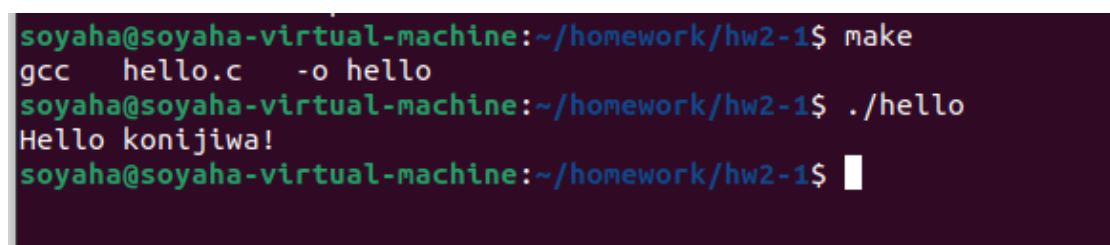
3. 用变量替换改写 Makefile 并验证



```
# 变量替换版Makefile
CC = gcc # 定义编译器变量
TARGET = hello # 定义目标文件变量
SRCS = hello.c # 定义源文件变量
HDRS = hello.h # 定义头文件变量

# 使用变量构建规则
$(TARGET): $(SRCS) $(HDRS)
    $(CC) $(SRCS) -o $(TARGET)

# 清理规则
clean:
    rm -f $TARGET
```



```
soyaha@soyaha-virtual-machine:~/homework/hw2-1$ make
gcc hello.c -o hello
soyaha@soyaha-virtual-machine:~/homework/hw2-1$ ./hello
Hello konijiwa!
soyaha@soyaha-virtual-machine:~/homework/hw2-1$
```

可以看到也是成功运行了

4. 写“无变量、双目标”的 Makefile1 并验证

```
soyaha@soyaha-virtual-machine: ~/homework/hw2-1
hello: hello.o
    gcc hello.o -o hello
hello.o: hello.c hello.h
    gcc -c hello.c -o hello.o
clean:
    rm -f hello hello.o
```

```
soyaha@soyaha-virtual-machine: ~/homework/hw2-1
soyaha@soyaha-virtual-machine:~/homework$ cd hw2-1/
soyaha@soyaha-virtual-machine:~/homework/hw2-1$ vi Makefile1
soyaha@soyaha-virtual-machine:~/homework/hw2-1$ make -f Makefile1
Makefile1:2: *** missing separator. Stop.
soyaha@soyaha-virtual-machine:~/homework/hw2-1$ make -f Makefile1
Makefile1:1: *** recipe commences before first target. Stop.
soyaha@soyaha-virtual-machine:~/homework/hw2-1$ make -f Makefile1
gcc -c hello.c -o hello.o
gcc hello.o -o hello
soyaha@soyaha-virtual-machine:~/homework/hw2-1$ ./hello
Hello konijiwa!
soyaha@soyaha-virtual-machine:~/homework/hw2-1$ make -f Makefile1 clean
rm -f hello hello.o
soyaha@soyaha-virtual-machine:~/homework/hw2-1$
```

经过验证可以发现采用这种方式编写的 makefile 执行 `make -f Makefile1`，会先生成 `hello.o`，再生成 `hello`

最后也是成功输出 `hello`

5. 用“变量替换”改写 Makefile1 并验证

进入 vim 改写 makefile1 的代码，如下：

```
soyaha@soyaha-virtual-machine: ~/homework/hw2-1
CC = gcc
TARGET = hello
OBJ = hello.o      # 用变量存中间目标文件
SRCS = hello.c
HDRS = hello.h
$(TARGET): $(OBJ)
    $(CC) $(OBJ) -o $(TARGET) # 链接命令（变量版）
$(OBJ): $(SRCS) $(HDRS)
    $(CC) -c $(SRCS) -o $(OBJ) # 编译命令（变量版）
clean:
    rm -f $(TARGET) $(OBJ)
~
~
~
~
~
~
~
-- INSERT --
```

编译运行

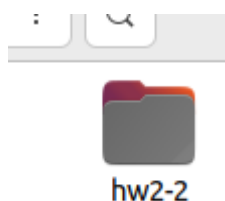
```
soyaha@soyaha-virtual-machine:~/homework/hw2-1$ vi Makefile1
soyaha@soyaha-virtual-machine:~/homework/hw2-1$ make -f Makefile1
gcc -c hello.c -o hello.o      # 编译命令（变量版）
gcc hello.o -o hello # 链接命令（变量版）
soyaha@soyaha-virtual-machine:~/homework/hw2-1$ ./hello
Hello konijiwa!
soyaha@soyaha-virtual-machine:~/homework/hw2-1$ make -f Makefile1 clean
rm -f hello hello.o
soyaha@soyaha-virtual-machine:~/homework/hw2-1$
```

可以发现结果也是正常的

作业 2-2:GDB 调试工具的使用

1. 创建错误代码文件 greet.c

先建一个新的文件夹



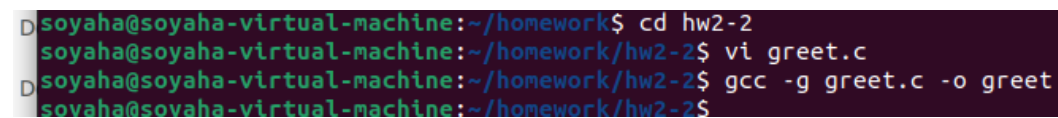
然后在终端用 vi 创建一个 greet.c 的文件



```
1#include <stdio.h>
2#include <string.h>
3#include <stdlib.h>
4
5// 函数声明
6int display1(char *string);
7int display2(char *string);
8
9int main(int argc, char **argv)
10{
11    char string[] = "Embedded Linux";
12    display1(string);
13    display2(string);
14    return 0;
15}
16
17// 打印原始字符串
18int display1(char *string)
19{
20    printf("The original string is %s\n", string);
21    return 0;
22}
23
24// 反转字符串并打印（修正后，解决内存越界和结束符问题）
25int display2(char *string1)
26{
27    char *string2;
28    int size, i;
29    size = strlen(string1);
30    // 分配足够内存（字符串长度 + 结束符 '\0'）
31    string2 = (char *)malloc(size + 1);
32    if (string2 == NULL) { // 内存分配失败时的错误处理（可选）
33        printf("Memory allocation failed!\n");
34        return -1;
35    }
36    // 逐个字符反转复制
37    for (i = 0; i < size; i++) {
38        //string2[i] = string1[size - 1 - i];
39        string2[size - i] = string1[i];
40    }
41    string2[size] = '\0'; // 为字符串添加结束符，确保合法
42    printf("The string afterward is %s\n", string2);
43    free(string2); // 释放动态分配的内存，避免内存泄漏
44    return 0;
45}
```

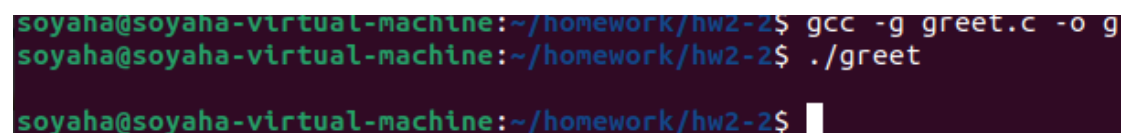
用这样一个有问题的代码来模拟代码有 bug 的情况

2.使用 gcc 编译这段代码



```
D soyaha@soyaha-virtual-machine:~/homework$ cd hw2-2
soyaha@soyaha-virtual-machine:~/homework/hw2-2$ vi greet.c
D soyaha@soyaha-virtual-machine:~/homework/hw2-2$ gcc -g greet.c -o greet
soyaha@soyaha-virtual-machine:~/homework/hw2-2$
```

3.运行生成的可执行文件./greet，观察运行结果。



```
soyaha@soyaha-virtual-machine:~/homework/hw2-2$ gcc -g greet.c -o g
soyaha@soyaha-virtual-machine:~/homework/hw2-2$ ./greet

soyaha@soyaha-virtual-machine:~/homework/hw2-2$
```

可以发现没有输出

4.使用 gdb 调试程序，通过设置断点、单步跟踪，一步步找出错误所在。

先进入 gdb 调试界面

```
soyaha@soyaha-virtual-machine:~/homework/hw2-2$ gdb greet
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from greet...
(gdb)
```

在开头设置断点

```
(gdb) break main
Breakpoint 1 at 0x1175: file greet.c, line 3.
(gdb)
```

然后单步执行

```
Breakpoint 1 at 0x1175: file greet.c, line 3.
(gdb) run
Starting program: /home/soyaha/homework/hw2-2/greet
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at greet.c:3
3   int main() {
(gdb) n
4   char str[] = "Hello";
(gdb)
```

执行到 display2 函数时，会看到字符串反转时的内存越界和结束符错误，导致程序可能行为异常（比如输出乱码、崩溃）

```
(gdb) n
15      }
(gdb) n
__libc_start_call_main (main=main@entry=0x55555555551e9 <main>, argc=argc@entry=1
, argv=argv@entry=0x7fffffffd8c8) at ../sysdeps/nptl/libc_start_call_main.h:74
74      ../sysdeps/nptl/libc_start_call_main.h: No such file or directory.
(gdb) n
[Inferior 1 (process 2991) exited normally]
```

5.纠正错误，更改源程序并得到正确的结果。

通过 vi 把 len 改成正确的

```
for (i = 0; i < size; i++) {
    string2[i] = string1[size - 1 - i];
    //string2[size-i] = string1[i];
}
```

内存越界：分配了 size+1 字节内存（用于存 size 个字符 + 字符串结束

符 \0）。但循环里写数据时，用了 string2[size - i]，最后还写

了 string2[size+1] 这超出了分配的内存范围（合法索引是 0 到 size），会导致“内存越界访问”。

字符串结束符错误：代码里写 string2[size+1] = "";，但 C 语言中字符串结束符是 \0，不是空字符；而且位置也越界了。

再编译运行

```
soyaha@soyaha-virtual-machine:~/homework/hw2-2$ gcc -g greet.c -o greet
soyaha@soyaha-virtual-machine:~/homework/hw2-2$ ./greet
The original string is Embedded Linux
The string afterward is xuniL deddebM
```

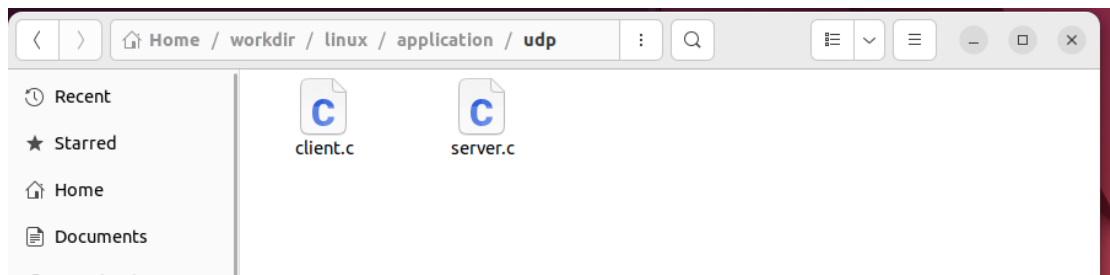
可以看到成功把字符串倒序输出了

作业 2-3：Linux 系统 UDP 网络协议编程

首先创建目录

```
soyaha@soyaha-virtual-machine: ~  
soyaha@soyaha-virtual-machine:~$ mkdir ~/workdir/linux/application/udp -p  
soyaha@soyaha-virtual-machine:~$
```

然后把 server 和 client 的文件复制进去



分别编译.c 文件，生成可执行文件 client 和 server

```
soyaha@soyaha-virtual-machine:~$ cd workdir/linux/application/udp/  
soyaha@soyaha-virtual-machine:~/workdir/linux/application/udp$ gcc server.c -o server  
soyaha@soyaha-virtual-machine:~/workdir/linux/application/udp$ gcc client.c -o client  
soyaha@soyaha-virtual-machine:~/workdir/linux/application/udp$
```

用 ifconfig 找到自己的 ip

```
soyaha@soyaha-virtual-machine:~/workdir/linux/application/udp$ ifconfig  
docker0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500  
    inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255  
    ether da:ed:56:4c:e2:8d txqueuelen 0 (Ethernet)  
    RX packets 0 bytes 0 (0.0 B)  
    RX errors 0 dropped 0 overruns 0 frame 0  
    TX packets 0 bytes 0 (0.0 B)  
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0  
  
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500  
    inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255  
    ether 08:00:27:2d:7a:11 txqueuelen 1000 (Ethernet)  
    RX packets 0 bytes 0 (0.0 B)  
    RX errors 0 dropped 0 overruns 0 frame 0  
    TX packets 0 bytes 0 (0.0 B)  
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

然后运行 server

```
soyaha@soyaha-virtual-machine:~/workdir/linux/application/udp$ ./server 172.17.0.1 8888
```

打开另一个终端运行 client


```
soyaha@soyaha-virtual-machine:~/workdir/linux/application/udp$ ./client 172.17.0.1 8888
>
```

再在 client 端发送任意消息

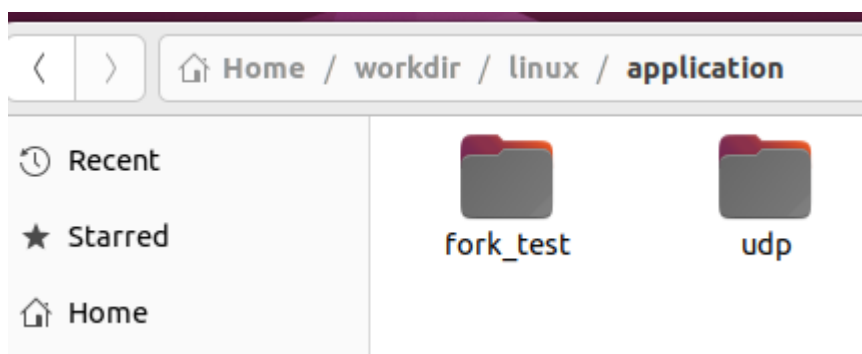
```
110.128
usage:./server ip port
soyaha@soyaha-virtual-machine:~/workdir/linux$ ./server 172.17.0.1 8888
from 192.168.110.128:46451 nihao
1 from 192.168.110.128:46451 hello
2
3
4#include <sys/types.h>
5#include <sys/socket.h>
5#include <errno.h>
7#include <string.h>
3#include <arpa/inet.h>
9#include <netinet/in.h>

.1
usage:./client ip port
soyaha@soyaha-virtual-machine:~/workdir/linux/application/udp$ ./client 172.17.0.1 8888
>nihao
nihao
>hello
hello
>
```

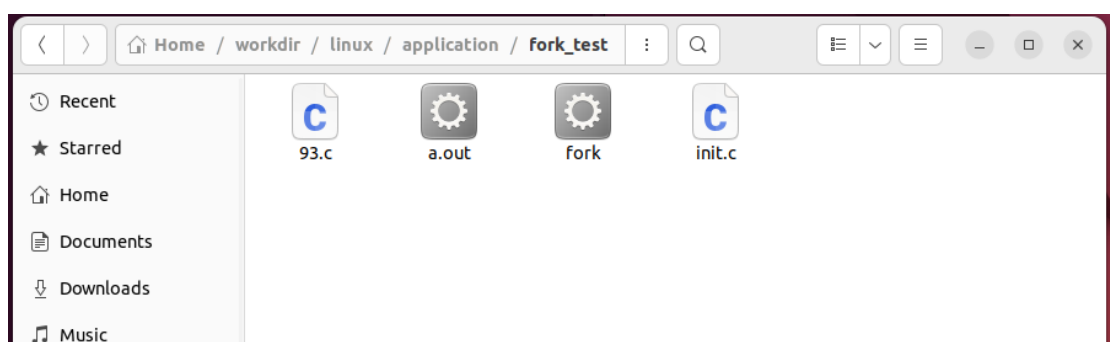
可以发现 server 端成功接收到了来自 client 端的消息

作业 2-4: linux 系统多进程实验

先创建出实验文件夹



把实验文件都复制进去



编译运行 93.c

```
soyaha@soyaha-virtual-machine:~$ cd workdir/linux/application/fork_test/  
soyaha@soyaha-virtual-machine:~/workdir/linux/application/fork_test$ ./fork  
bash: ./fork: Permission denied  
soyaha@soyaha-virtual-machine:~/workdir/linux/application/fork_test$ gcc 93.c -o  
fork  
soyaha@soyaha-virtual-machine:~/workdir/linux/application/fork_test$ ./fork  
the test content!  
fork test!  
global=24 test=2 Parent,my PID is 3401  
global=23 test=1 Child,my PID is 3402  
soyaha@soyaha-virtual-machine:~/workdir/linux/application/fork_test$
```

可以发现父子 PID 不同，父进程 PID 小于子进程 PID，因为子进程由父进程创建

由此可以得出结论：

- ① fork()创建的子进程是父进程的独立副本，**变量不共享**（修改互不影响）。
- ② 父子进程的**调度顺序由操作系统决定**，具有随机性。
- ③ fork()的返回值是区分父子进程的关键（父进程得子 PID，子进程得 0）。

心得体会

完成这组 Linux 系统编程作业，我在多方面收获了实践认知：Makefile 让我体会到工程化编译的便捷与可维护性，从基础编写到变量替换、多目标编译，清晰理解了依赖与增量编译逻辑；GDB 调试让我学会用专业工具高效定位代码 bug；UDP 网络编程亲手实现了网络通信，掌握了 Socket、网络字节序等关键

细节；多进程实验通过`fork`和变量变化，直观认识了写时复制与进程调度特性。这些实践层层递进，不仅构建了系统编程知识体系，也让我更敬畏 Linux 系统编程的细节，渴望探索更深层机制。