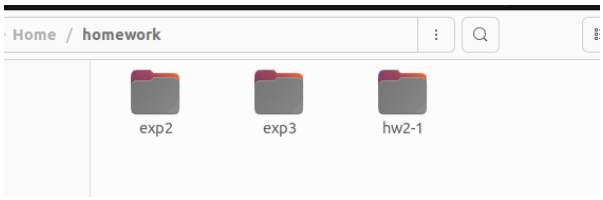
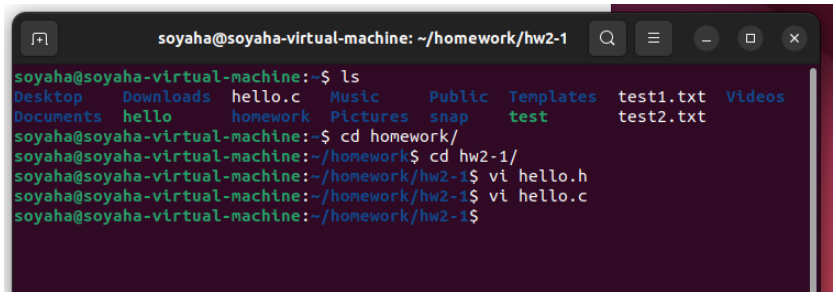
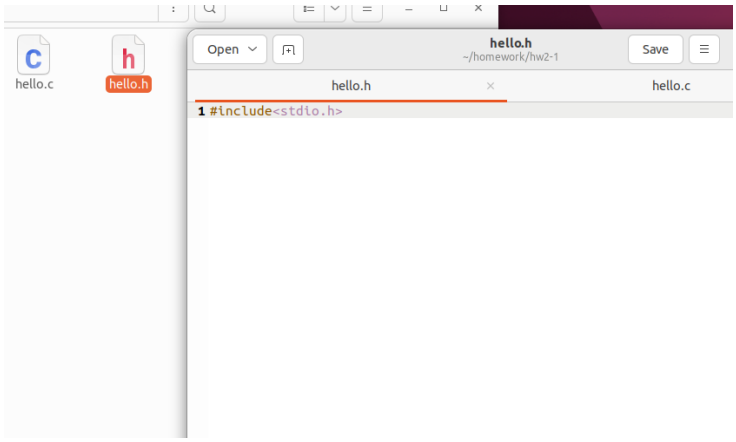
作业 2-1:makefile 工程管理器的使用

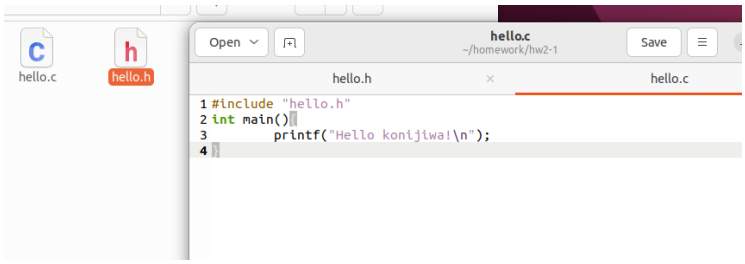
先提前创建好作业文件夹



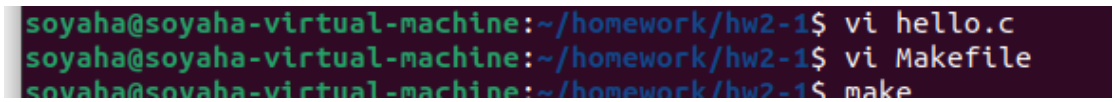
1.然后使用 vi 编辑出 hello.c 和 hello.h 文件

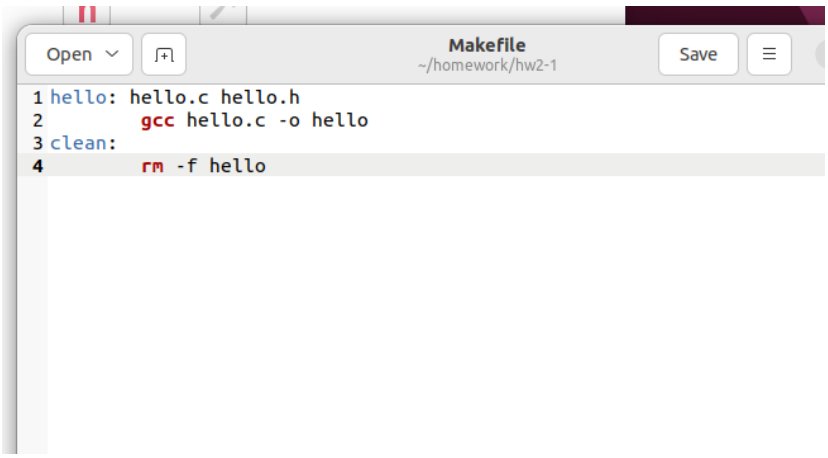


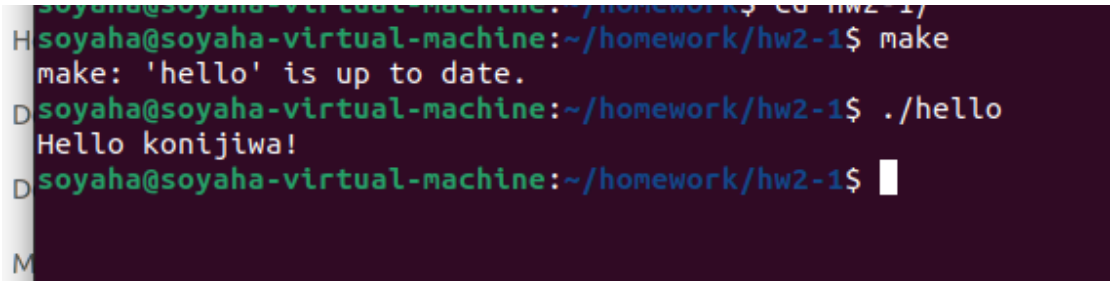




2.再编写无变量、单目标的 Makefile 文件并验证

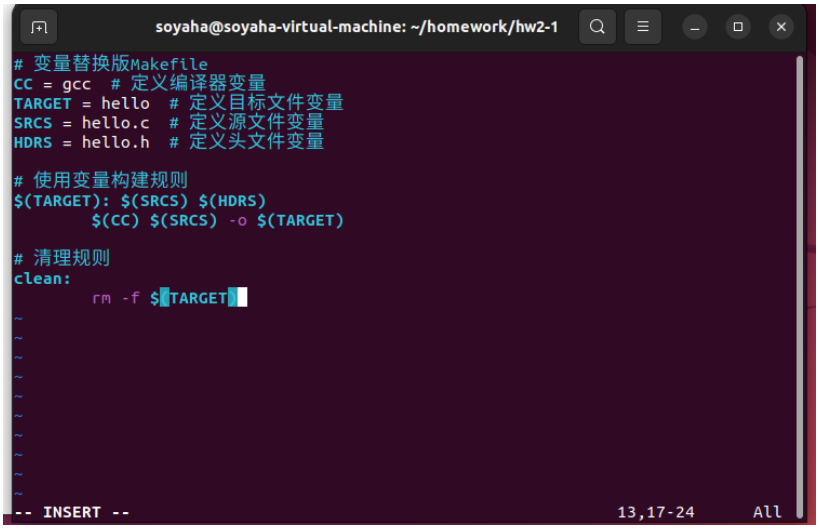


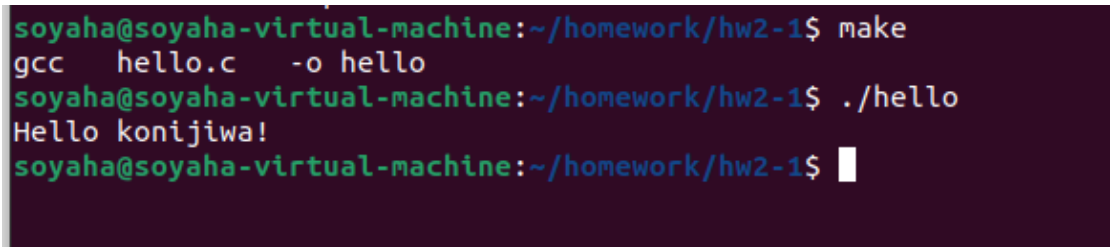




可以发现成功编译运行了 hello 程序

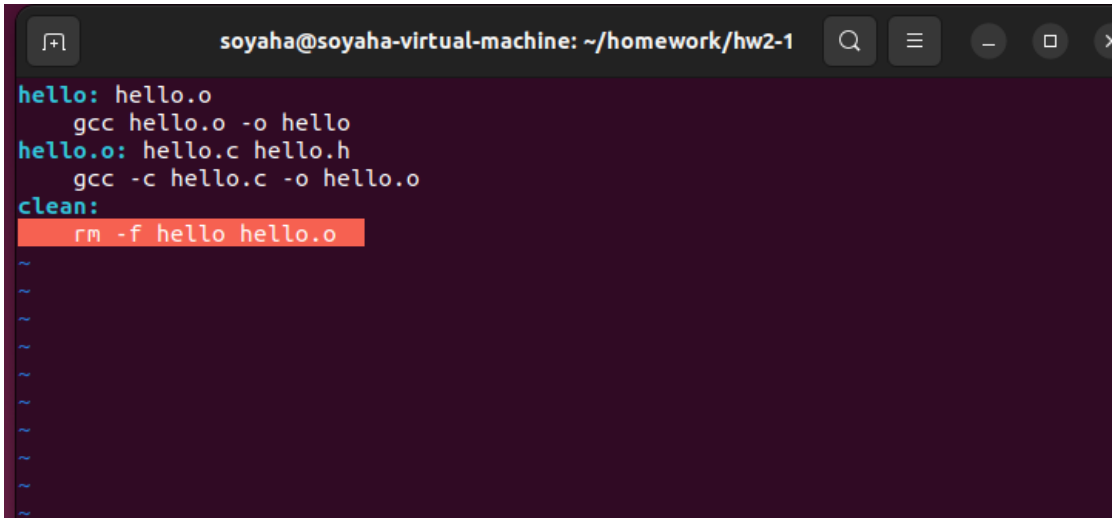
3. 用变量替换改写 Makefile 并验证

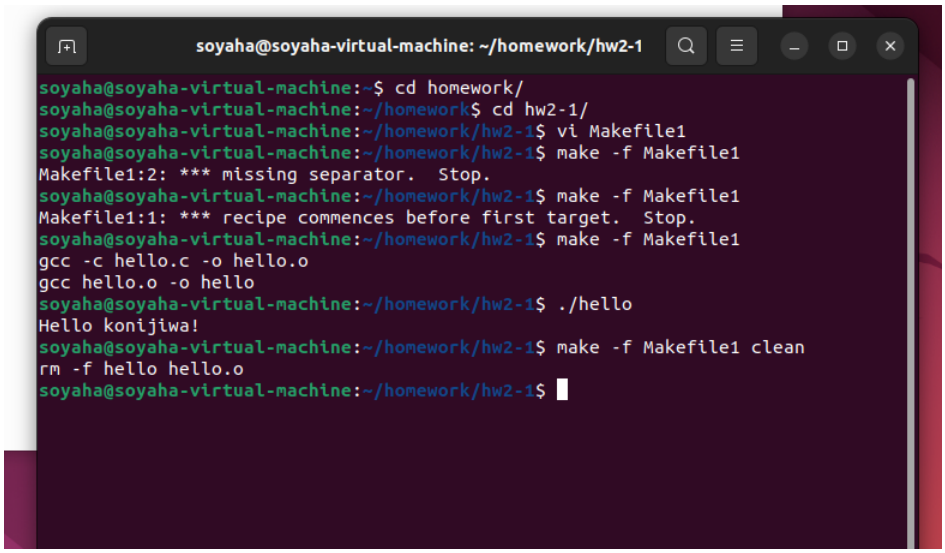




可以看到也是成功运行了

4. 写“无变量、双目标”的 Makefile1 并验证





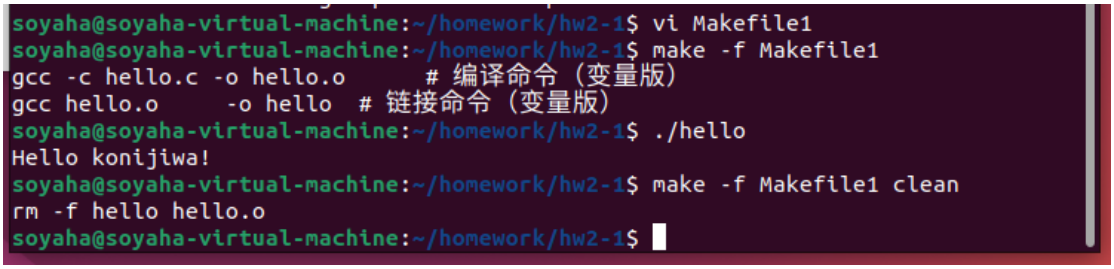
经过验证可以发现采用这种方式编写的 makefile 执行 make -f Makefile1，会先生成 hello.o，再生成 hello

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：

编译运行

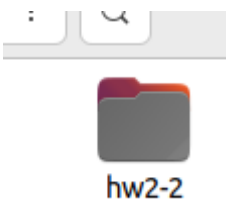


可以发现结果也是正常的

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

Open greet.c ~/homework/hw2-2

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// 函数声明
int display1(char *string);
int display2(char *string);

int main(int argc, char **argv)
{
    char string[] = "Embedded Linux";
    display1(string);
    display2(string);
    return 0;
}

// 打印原始字符串
int display1(char *string)
{
    printf("The original string is %s\n", string);
    return 0;
}

// 反转字符串并打印（修正后，解决内存越界和结束符问题）
int display2(char *string1)
{
    char *string2;
    int size, i;
    size = strlen(string1);
    // 分配足够内存（字符串长度 + 结束符 '\0'）
    string2 = (char *)malloc(size + 1);
    if (string2 == NULL) { // 内存分配失败时的错误处理（可选）
        printf("Memory allocation failed!\n");
        return -1;
    }

    // 逐个字符反转复制
    for (i = 0; i < size; i++) {
        //string2[i] = string1[size - 1 - i];
        string2[size-i] = string1[i];
    }
    string2[size] = '\0'; // 为字符串添加结束符，确保合法
    printf("The string afterward is %s\n", string2);
    free(string2); // 释放动态分配的内存，避免内存泄漏
    return 0;
}
```

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

```
soyaha@soyaha-virtual-machine:~/homework$ cd hw2-2
soyaha@soyaha-virtual-machine:~/homework/hw2-2$ vi greet.c
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。

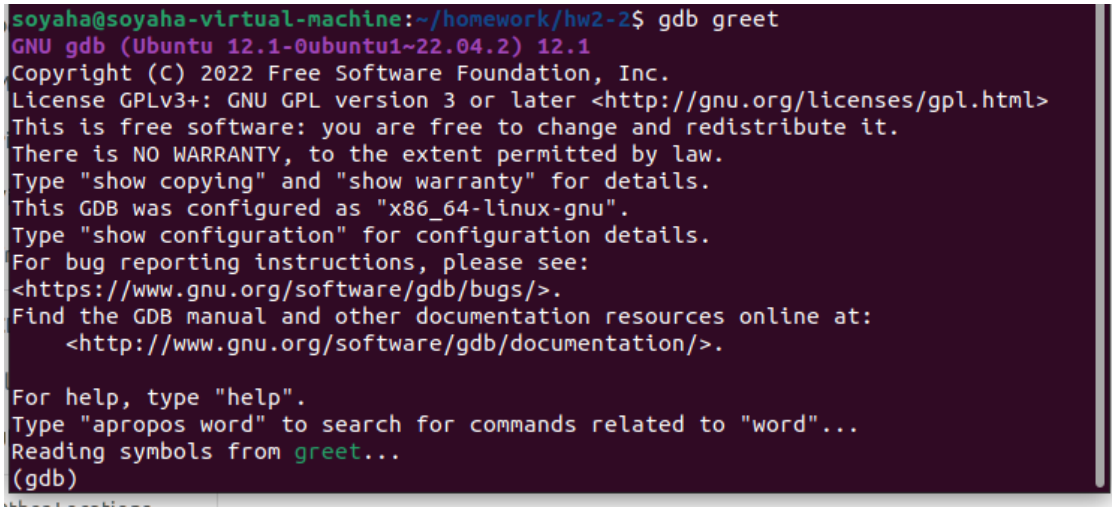
```
soyaha@soyaha-virtual-machine:~/homework/hw2-2$ gcc -g greet.c -o g
soyaha@soyaha-virtual-machine:~/homework/hw2-2$ ./greet

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

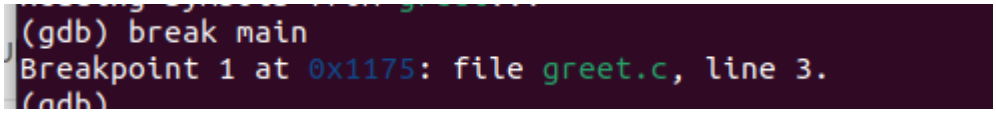
可以发现没有输出

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。

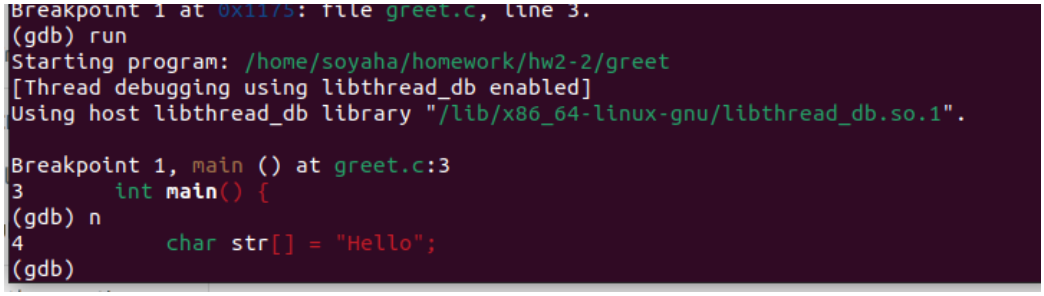
先进入 gdb 调试界面



在开头设置断点



然后单步执行



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=0x5555555551e9 <main>, argc=argc@entry=1
, argv=argv@entry=0x7fffffffdfc8) 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。

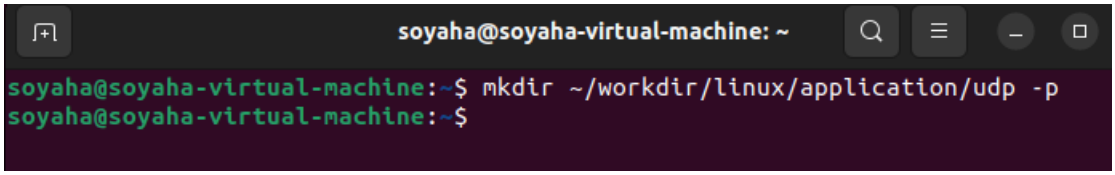
再编译运行

```
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 deddebmE
```

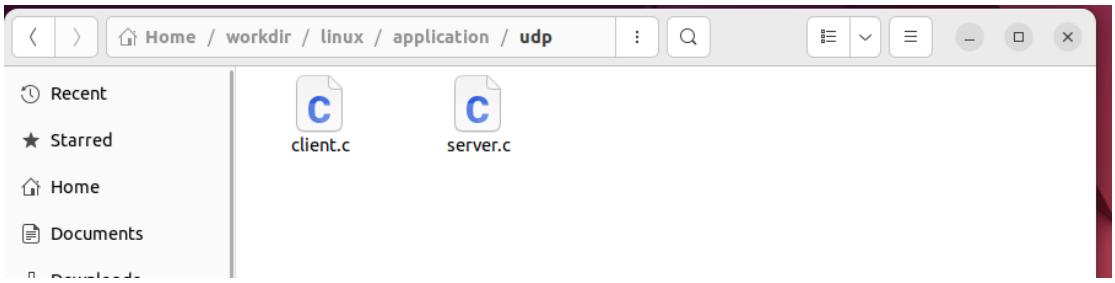
可以看到成功把字符串倒序输出了

作业 2-3：Linux 系统 UDP 网络协议编程

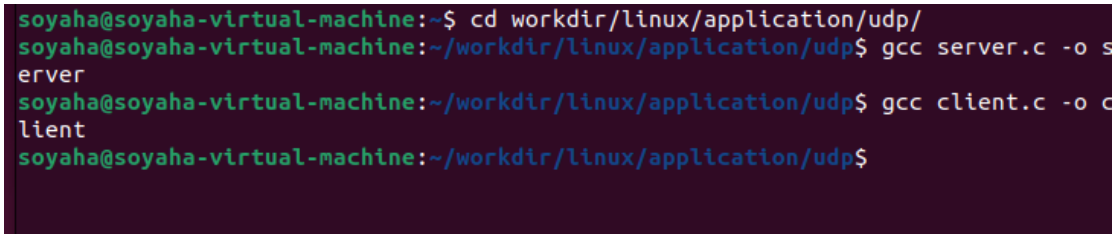
首先创建目录



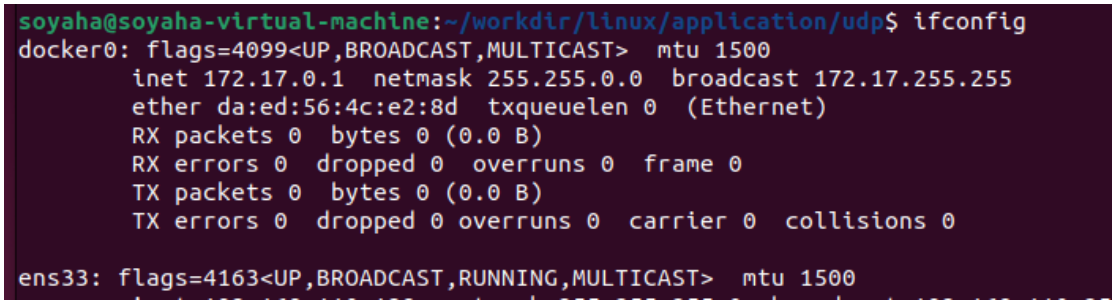
然后把 server 和 client 的文件复制进去



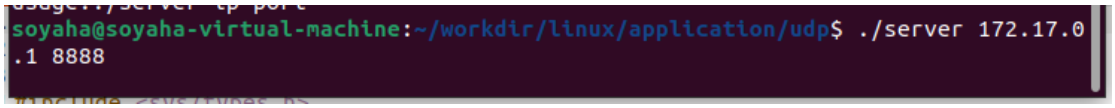
分别编译.c 文件，生成可执行文件 client 和 server



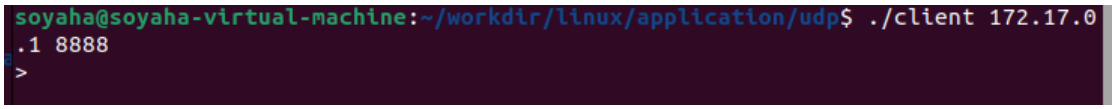
用 ifconfig 找到自己的 ip



然后运行 server



打开另一个终端运行 client

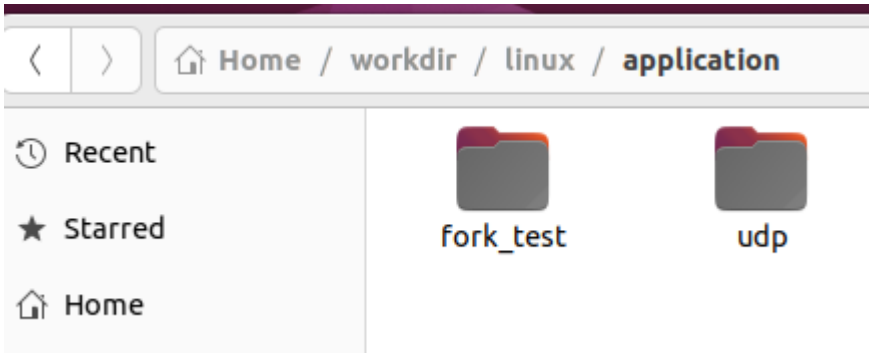


再在 client 端发送任意消息

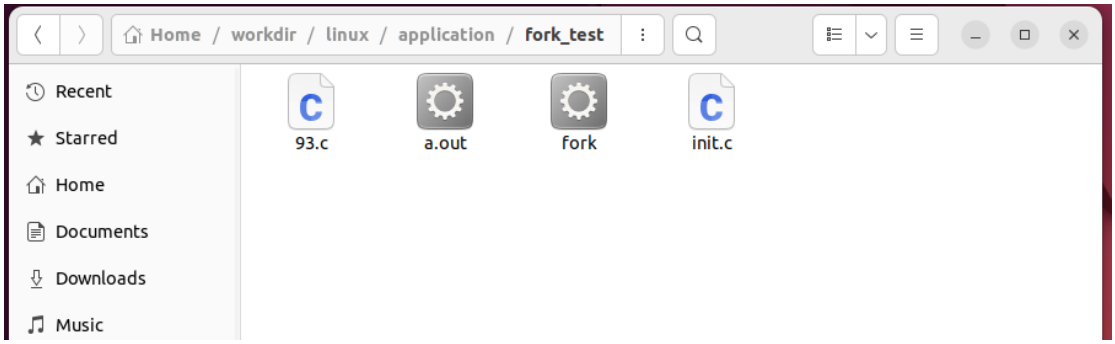
可以发现 server 端成功接收到了来自 client 端的消息

作业 2-4：linux 系统多进程实验

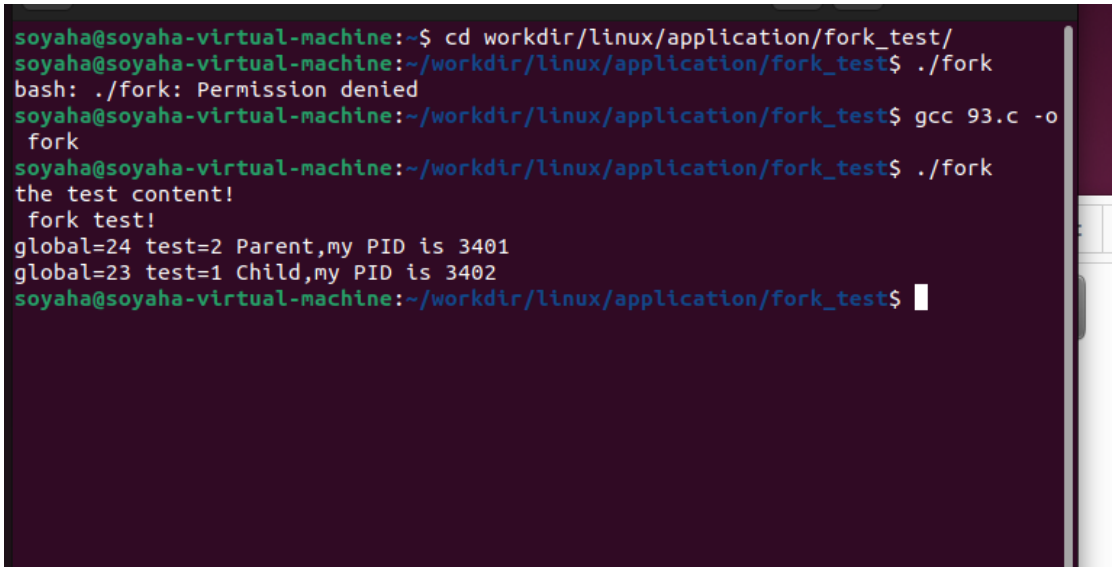
先创建出实验文件夹



把实验文件都复制进去



编译运行 93.c



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。