

1、什么是 linux 终端？有哪几种类型？

参考资料：

<https://waynerv.com/posts/how-tty-system-works/>

答：

（1）Linux 终端是用户与 Linux 操作系统进行交互的界面。它允许用户通过命令行输入指令来执行各种操作，例如管理文件、运行程序、配置系统等。Linux 终端也称为命令行界面。

（2）在 Linux 系统中，有以下几种类型的终端：虚拟终端、终端仿真器、远程终端、串行端口终端、伪终端、控制终端、控制台终端

2、读入键盘的编码。

理解原始（nonblock）输入。

```
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <fcntl.h>

int main()
{
    // 保存终端属性
    struct termios old_attrs, new_attrs;
    unsigned char c_in[4];
    int c;
    int ret;
    int attr = 0;

    tcgetattr(STDIN_FILENO, &old_attrs);

    // 设置终端属性
    new_attrs = old_attrs;
    new_attrs.c_cc[VTIME] = 0; // 等待的超时时间 0。1s 的整数，0 表示 non block
    new_attrs.c_cc[VMIN] = 0; // 等待读到的数量，

    new_attrs.c_lflag &= ~(ICANON | ECHO);
    tcsetattr(STDIN_FILENO, TCSANOW, &new_attrs);

    // 打印提示信息
    printf("Press any key. Press 'q' to quit.\n");

    // 读取并打印按键值
```

```

while (1) {
//  char c = getchar();
//  printf("Key: %c (%d) code:%02x\n", c, c,c);
c_in[0]=0;
c_in[1]=0;
c_in[2]=0;
    ret = read(STDIN_FILENO, c_in, 3);
    if(ret) printf("Key : %02x , %02x , %02x  RET:%d\n", c_in[0], c_in[1],c_in[2],ret);
        if (c_in[0] == 'q') { // 处理退出
            break;
        }
// printf("ret:%d\n",ret);
}
// 恢复终端属性
tcsetattr(STDIN_FILENO, TCSANOW, &old_attrs);

return 0;
}

```

将以上程序命名为 key.c

```
gcc key.c
```

```
./a.out
```

按上，下，左，右方向键，看程序的输出？

输入中文 严 字，看程序的输出？

```

USER2022102147@localhost:/mnt/c/Users/洗胡兵$ vi key.c
USER2022102147@localhost:/mnt/c/Users/洗胡兵$ gcc key.c

```

```

USER2022102147@localhost:/mnt/c/Users/洗胡兵$ ./a.out
Press any key. Press 'q' to quit.
Key : 1b , 5b , 41  RET:3
Key : 1b , 5b , 42  RET:3
Key : 1b , 5b , 44  RET:3
Key : 1b , 5b , 43  RET:3
Key : e4 , b8 , a5  RET:3

```

参考文档：

<https://www.cnblogs.com/mmxingye/p/16296138.html>

<https://gist.github.com/fnky/458719343aabd01cfb17a3a4f7296797>

3. 通过 ncurses 库，实现方向键及光标的位置控制。

<https://tldp.org/HOWTO/NCURSES-Programming-HOWTO/keys.html>

安装 NCURSES 库

```
sudo apt-get install libncurses5-dev libncursesw5-dev
```

```
#include <stdio.h>
```

```

#include <ncurses.h>

#define WIDTH 30
#define HEIGHT 10

int startx = 0;
int starty = 0;

char *choices[] = {
    "Choice 1",
    "Choice 2",
    "Choice 3",
    "Choice 4",
    "Exit",
};

int n_choices = sizeof(choices) / sizeof(char *);
void print_menu(WINDOW *menu_win, int highlight);

int main()
{
    WINDOW *menu_win;
    int highlight = 1;
    int choice = 0;
    int c;

    initscr();
    clear();
    noecho();
    cbreak(); /* Line buffering disabled. pass on everything */
    startx = (80 - WIDTH) / 2;
    starty = (24 - HEIGHT) / 2;

    menu_win = newwin(HEIGHT, WIDTH, starty, startx);
    keypad(menu_win, TRUE);
    mvprintw(0, 0, "Use arrow keys to go up and down, Press enter to select a choice");
    refresh();
    print_menu(menu_win, highlight);
    while(1)
    {
        c = wgetch(menu_win);
        switch(c)
        {
            case KEY_UP:
                if(highlight == 1)
                    highlight = n_choices;
                else
                    --highlight;

```

```

        break;
    case KEY_DOWN:
        if(highlight == n_choices)
            highlight = 1;
        else
            ++highlight;
        break;
    case 10:
        choice = highlight;
        break;
    default:
        mvprintw(24, 0, "Charcter pressed is = %3d Hopefully it can be printed as
'%c'", c, c);

        refresh();
        break;
    }
    print_menu(menu_win, highlight);
    if(choice != 0) /* User did a choice come out of the infinite loop */
        break;
}
mvprintw(23, 0, "You chose choice %d with choice string %s\n", choice, choices[choice - 1]);
clrtoeol();
refresh();
endwin();
return 0;
}

```

```

void print_menu(WINDOW *menu_win, int highlight)
{
    int x, y, i;

    x = 2;
    y = 2;
    box(menu_win, 0, 0);
    for(i = 0; i < n_choices; ++i)
    {
        if(highlight == i + 1) /* High light the present choice */
        {
            watttrn(menu_win, A_REVERSE);
            mvwprintw(menu_win, y, x, "%s", choices[i]);
            wattroff(menu_win, A_REVERSE);
        }
        else
            mvwprintw(menu_win, y, x, "%s", choices[i]);
        ++y;
    }
}

```

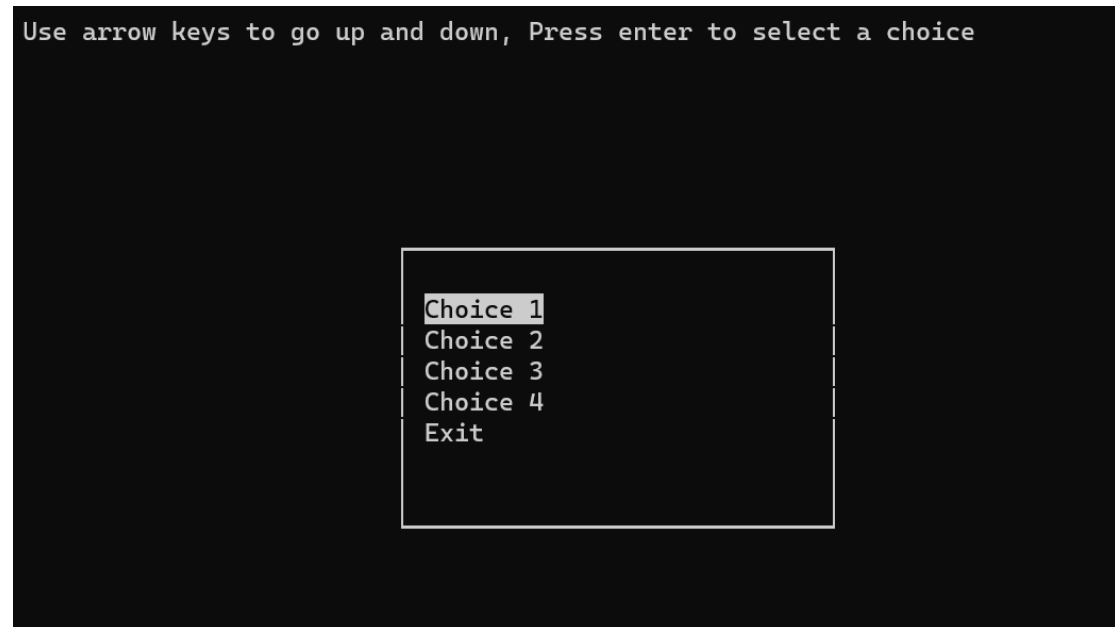
```

    }
    wrefresh(menu_win);
}

```

将以上程序命令为 keys.c

gcc keys.c -lcurses -o keys #带 curses 库编译，生成 keys 程序，-lcurses 表示需要库 curses  
./keys # 运行程序



2).贪吃蛇游戏:

[https://blog.csdn.net/weixin\\_53762042/article/details/122015819](https://blog.csdn.net/weixin_53762042/article/details/122015819)

gcc snake.c -lpthread -lcurses -o snake

./snake

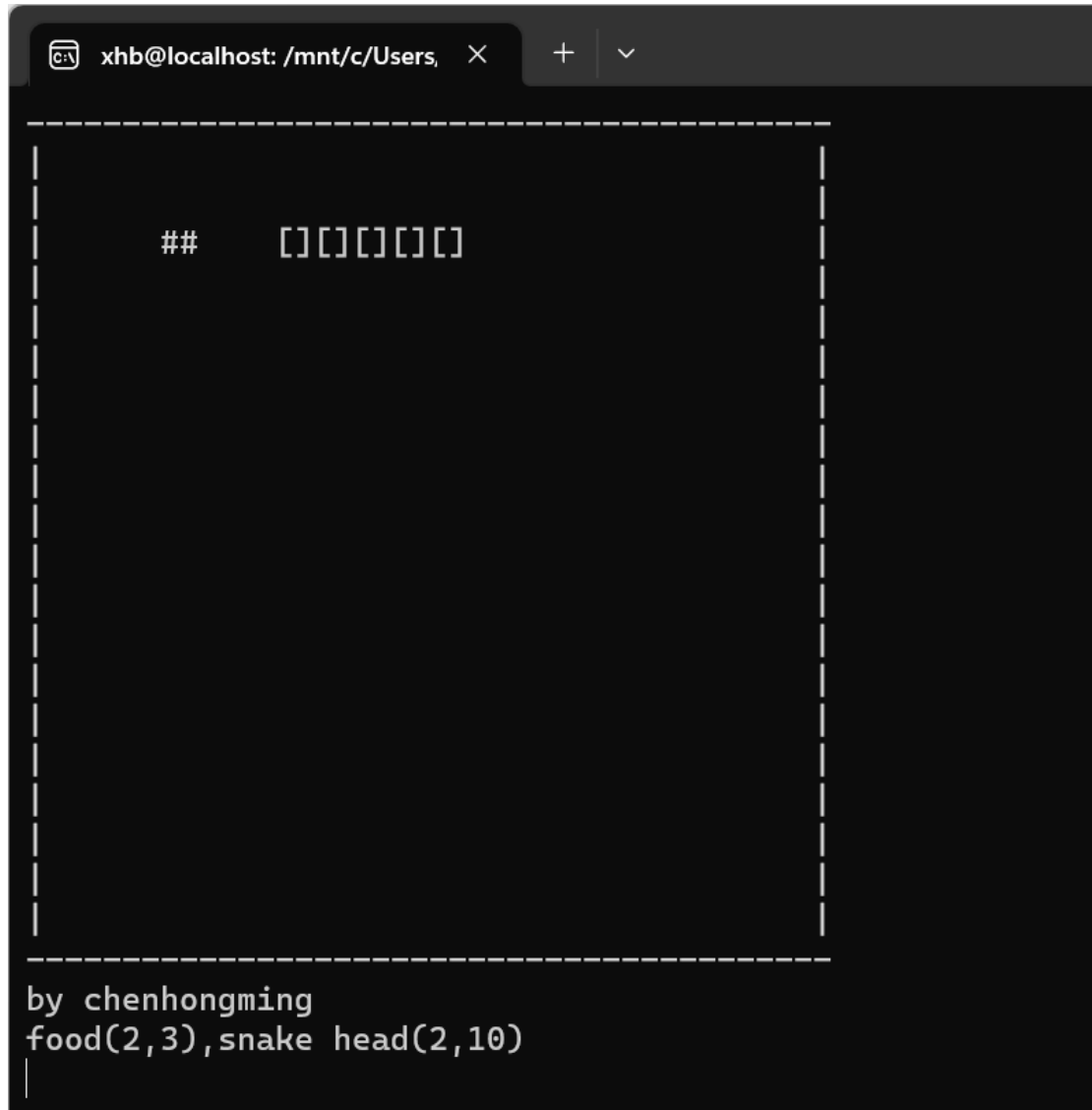
其中移动的速度通过以下的延迟来控制调节:

```
usleep(10000); // 213 行，调节速度，加大延迟，可减慢速度
```

```

xhb@localhost:/mnt/c/Users/洗胡兵$ vi snake.c
xhb@localhost:/mnt/c/Users/洗胡兵$ gcc snake.c -lpthread -lcurses -o snake
xhb@localhost:/mnt/c/Users/洗胡兵$ ./snake

```



4、<https://www.cnblogs.com/lingyejun/p/8312838.html>

参考以上网页，理解 LSB、MSB 和大小端模式及网络字节序

```
#include <stdio.h>
```

```
char str[]="hello, 1234";
```

```
int value1=0x1234;
```

```
int value2=-3;
```

```
char *ptr;
```

```
void main()
```

```
{
```

```
int j, sizeofint;
```

```
sizeofint=sizeof(int);
```

```
printf("str=%s \n",str);
```

```
printf("value1 hex:%0x dec:%d\n",value1,value1);
```

```

ptr=(char *) &value1;
printf("%p \n",str);
for(j=0;j<sizeofint;j++) {
    printf("address:%p value:%02X\n",ptr,*ptr);
    ptr++;
}

printf("value2 hex:%0x dec:%d\n",value2,value2);
ptr=(char *) &value2;
for(j=0;j<sizeofint;j++) {
    printf("address:%p value:%02X\n",ptr,(unsigned char)*ptr);
    ptr++;
}

}

```

将以上程序 命名为 test-end.c, 运行以下命令, 解释命令的执行结果? 说明你的系统的大小端方式?

```

gcc test-end.c -o test-end
./test-end

```

```

xhb@localhost:/mnt/c/Users/洗胡兵$ vi test-end.c
xhb@localhost:/mnt/c/Users/洗胡兵$ gcc test-end.c -o test-end
xhb@localhost:/mnt/c/Users/洗胡兵$ ./test-end
str=hello, 1234
value1 hex:1234 dec:4660
0x55fe83221010
address:0x55fe8322101c value:34
address:0x55fe8322101d value:12
address:0x55fe8322101e value:00
address:0x55fe8322101f value:00
value2 hex:fffffffd dec:-3
address:0x55fe83221020 value:FD
address:0x55fe83221021 value:FF
address:0x55fe83221022 value:FF
address:0x55fe83221023 value:FF
xhb@localhost:/mnt/c/Users/洗胡兵$

```

解释:

- (1) 对于 `value1`: 其十六进制值为 `0x1234`, 在内存中的存储顺序为 `34 12 00 00`。这说明系统采用小端序 (Little Endian), 即最低有效字节 (低字节) 存储在最低内存地址处。
- (2) 对于 `value2`: 其十六进制值为 `0xffffffff`, 在内存中的存储顺序为 `FD FF FF FF`。这同样说明系统采用小端序 (Little Endian)。
- (3) 因此, 根据程序的输出, 系统采用小端序 (Little Endian) 方式存储数据。

## 5、将内存数据写入磁盘文件

<https://www.runoob.com/cprogramming/c-strings.html>

C 字符串表示

<https://www.cnblogs.com/andy-0212/p/10323502.html>

负数的二进制表示方法 (正数: 原

码、负数: 补码)

<http://t.csdn.cn/NkTOP> 浮点数的存储方式

<https://blog.csdn.net/Djsnxbjans/article/details/127283009> 文件相关概念

参考以上网页，理解下面程序：

```
#include <stdio.h>
```

```
char str[]="hello, 1234";
```

```
int value1 = 0x1234;
```

```
int value2 = -3;
```

```
float value3 =-10.5;
```

```
float value4 = 16.625;
```

```
float f1 = 123.234567;
```

```
float f2 = 123.234568;
```

```
int main()
```

```
{
```

```
int j, sizeofint;
```

```
FILE *fp;
```

```
if( (fp=fopen("fdata.txt", "wb+")) == NULL ){ //以二进制方式打开
```

```
puts("Fail to open file!");
```

```
return(-1);
```

```
}
```

```
sizeofint = sizeof(int);
```

```
fwrite(str, 12, 1, fp); // write 12 bytes;
```

```
fwrite(&value1, sizeof(int), 1, fp); // write int;
```

```
fwrite(&value2, sizeof(int), 1, fp); // write int ;
```

```
fwrite(&value3, sizeof(float), 1, fp); // write float ;
```

```
fwrite(&value4, sizeof(float), 1, fp); // write float ;
```

```
fwrite(&f1, sizeof(float), 1, fp); // write float ;
```

```
fwrite(&f2, sizeof(float), 1, fp); // write float ;
```

```
printf("str:%s \n", str);
```

```
printf("value1:%d \n", value1);
```

```
printf("value2:%d \n", value2);
```

```
printf("value3:%f \n", value3);
```

```
printf("value4:%f \n", value4);
```

```
printf("f1:%f \n", f1);
```

```
printf("f2:%f \n", f2);
```

```
fclose(fp);
```

```
return(0);
```



```
}
```

1)、将以上程序命名为 `fwrite.c`, 对以上程序的主要语句加上注解,

```
#include <stdio.h>

// 声明一些全局变量: 字符串、整型数、浮点数
char str[]="hello, 1234";
int value1 = 0x1234;
int value2 = -3;
float value3 = -10.5;
float value4 = 16.625;
float f1 = 123.234567;
float f2 = 123.234568;

int main() {
    int j, sizeofint;
    FILE *fp; // 文件指针

    // 尝试以二进制写入+读取模式打开文件 "fdata.txt"
    if ((fp=fopen("fdata.txt", "wb+")) == NULL) {
        puts("Fail to open file!"); // 如果打开文件失败, 打印错误消息并退出程序
        return(-1);
    }

    // 写入数据到文件
    fwrite(str, 12, 1, fp); // 写入字符串 str 的前 12 个字节
    fwrite(&value1, sizeof(int), 1, fp); // 写入整型数 value1
    fwrite(&value2, sizeof(int), 1, fp); // 写入整型数 value2
    fwrite(&value3, sizeof(float), 1, fp); // 写入浮点数 value3
    fwrite(&value4, sizeof(float), 1, fp); // 写入浮点数 value4
    fwrite(&f1, sizeof(float), 1, fp); // 写入浮点数 f1
    fwrite(&f2, sizeof(float), 1, fp); // 写入浮点数 f2

    // 通过标准输出打印变量的值
    printf("str:%s \n", str);
    printf("value1:%d \n", value1);
    printf("value2:%d \n", value2);
    printf("value3:%f \n", value3);
    printf("value4:%f \n", value4);
    printf("f1:%f \n", f1);
    printf("f2:%f \n", f2);

    fclose(fp); // 关闭文件
    return(0);
}
```

2)、运行以下命令, 解释命令的执行结果?

```
gcc fwrite.c -o fwrite
```

```
./fwrite
```

```
ls -l fdata.txt
```

```
cat fdata.txt
```

```
xxd fdata.txt
```



```
xhb@localhost: /mnt/c/Users, × + v
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *fp;
    char str[13]; // 额外一个字节用于 '\0', 以确保字符串正确终止
    int value1, value2;
    float value3, value4, f1, f2;

    // 以二进制读取模式打开文件
    fp = fopen("fdata.txt", "rb");
    if (fp == NULL) {
        perror("Error opening file");
        return(-1);
    }

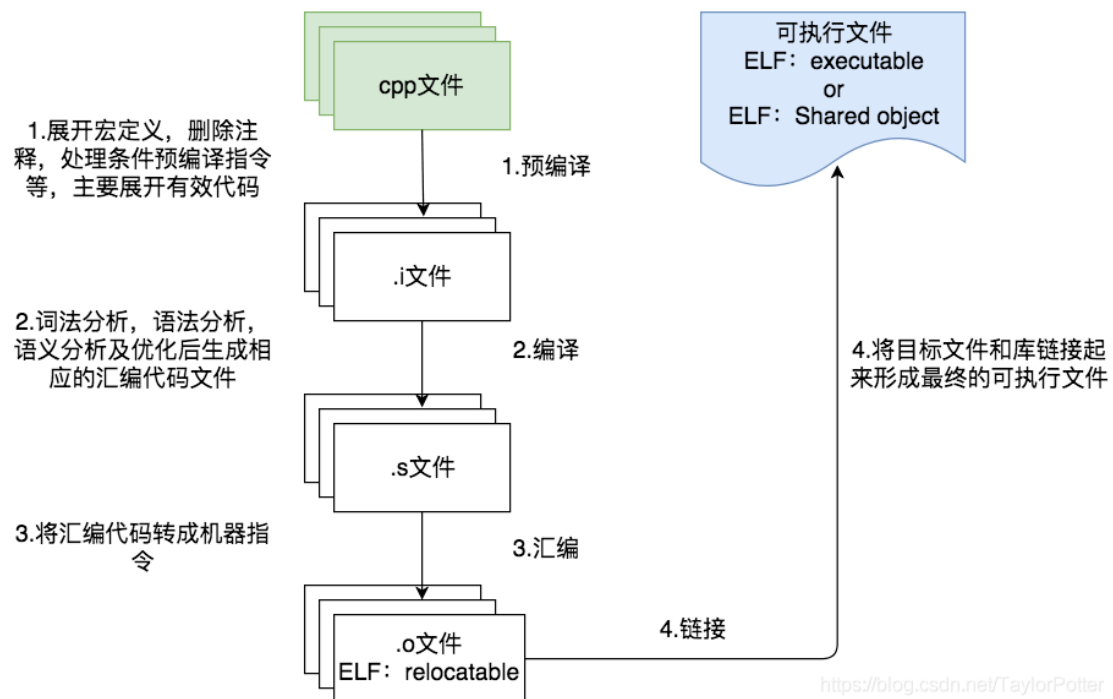
    // 读取数据
    fread(str, 12, 1, fp); // 读取字符串
    str[12] = '\0'; // 确保字符串正确终止
    fread(&value1, sizeof(int), 1, fp); // 读取 int value1
    fread(&value2, sizeof(int), 1, fp); // 读取 int value2
    fread(&value3, sizeof(float), 1, fp); // 读取 float value3
    fread(&value4, sizeof(float), 1, fp); // 读取 float value4
    fread(&f1, sizeof(float), 1, fp); // 读取 float f1
    fread(&f2, sizeof(float), 1, fp); // 读取 float f2

    // 打印读取的数据
    printf("str: %s\n", str);
    printf("value1: %d\n", value1);
    printf("value2: %d\n", value2);
    printf("value3: %f\n", value3);
    printf("value4: %f\n", value4);
    printf("f1: %f\n", f1);
    printf("f2: %f\n", f2);

    // 关闭文件
    fclose(fp);

    return 0;
}
```

## 7、了解 linux 可执行程序（ELF 格式文件）的结构



[https://blog.csdn.net/weixin\\_44966641/article/details/120328488](https://blog.csdn.net/weixin_44966641/article/details/120328488)

ELF 文件详解—初步认识

<https://zhuanlan.zhihu.com/p/30516267>

readelf 命令和 ELF 文件详解

<https://www.cnblogs.com/yizhanwillsucceed/p/13578076.html>

linux 进程地址空间划分

<https://www.cnblogs.com/coderkian/p/3840582.html>

linux 下汇编语言开发总结

<http://www.caiyunlin.com/2021/07/assembly-primer/>

汇编语言基础

1)、将下面文件保存为 **main.c**

```
int main(){
    return 0;
}
```

运行以下命令，给出命令结果，并解释命令的功能

```
$ gcc -g -S -o main.s main.c
$ cat main.s
$ gcc -o main main.s
$ ./main
$ echo $?
```

```

xhb@localhost:/mnt/c/Users/洗胡兵$ gcc -g -S -o main.s main.c
xhb@localhost:/mnt/c/Users/洗胡兵$ cat main.s
    .file    "main.c"
    .text
.Ltext0:
    .file 0 "/mnt/c/Users/洗胡兵/345\206\274\350\203\241\345\205\265" "main.c"
    .globl  main
    .type   main, @function
main:
.LFB0:
    .file 1 "main.c"
    .loc 1 1 11
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    .loc 1 2 9
    movl    $0, %eax
    .loc 1 3 1
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
.Letext0:
    .section        .debug_info,"",@progbits
.Ldebug_info0:
    .long     0x50
    .value    0x5
    .byte     0x1
    .byte     0x8
    .long     .Ldebug_abbrev0
    .uleb128   0x1
    .long     .LASF2
    .byte     0x1d
    .long     .LASF0
    .long     .LASF1
    .quad     .Ltext0
    .quad     .Letext0-.Ltext0
    .long     .Ldebug_line0
    .uleb128   0x2
    .long     .LASF3

```

```

        .byte    0
        .section      .debug_aranges,"",@progbits
        .long    0x2c
        .value   0x2
        .long    .Ldebug_info0
        .byte    0x8
        .byte    0
        .value   0
        .value   0
        .quad    .Ltext0
        .quad    .Ltext0-.Ltext0
        .quad    0
        .quad    0
        .section      .debug_line,"",@progbits
.Ldebug_line0:
        .section      .debug_str,"MS",@progbits,1
.LASF2:
        .string "GNU C17 11.4.0 -mtune=generic -march=x86-64 -g -fasynchronous-unwind-ta
.LASF3:
        .string "main"
        .section      .debug_line_str,"MS",@progbits,1
.LASF1:
        .string "/mnt/c/Users/\345\206\274\350\203\241\345\205\265"
.LASF0:
        .string "main.c"
        .ident "GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0"
        .section      .note.GNU-stack,"",@progbits
        .section      .note.gnu.property,"a"
        .align 8
        .long    1f - 0f
        .long    4f - 1f
        .long    5
0:
        .string "GNU"
1:
        .align 8
        .long    0xc0000002
        .long    3f - 2f
2:
        .long    0x3
3:
        .align 8
4:
xhb@localhost:/mnt/c/Users/洗胡兵$ gcc -o main main.s
xhb@localhost:/mnt/c/Users/洗胡兵$ ./main
xhb@localhost:/mnt/c/Users/洗胡兵$ echo $?
0
xhb@localhost:/mnt/c/Users/洗胡兵$

```

解释功能:

## 1. gcc -g -S -o main.s main.c

`gcc` 是 GNU Compiler Collection (GNU 编译器集合) 的一部分, 用于编译 C 语言文件。

`-g` 选项告诉编译器添加调试信息到输出文件中, 这对于调试程序非常有用。

`-S` 选项告诉编译器只进行到编译阶段, 停止在汇编阶段。这将生成一个汇编语言文件。

`-o main.s` 指定输出文件的名称为 `main.s`。

`main.c` 是输入的 C 语言源代码文件。

执行这个命令后，不会有直接的命令行输出，但它会生成一个名为 ``main.s`` 的文件，包含源代码 ``main.c`` 的汇编语言版本。

## 2. ``$ cat main.s``

- ``cat`` 是 `concatenate`（连接）的缩写，用于在标准输出上显示文件内容。
- 执行此命令将会在命令行上显示 ``main.s`` 文件的内容，即 ``main.c`` 源文件的汇编语言表示。

由于这是一个非常简单的程序，所以生成的汇编代码也会相对简单，内容会依赖于系统架构（比如 `x86_64` 或 `arm` 等），可能包括程序入口点、返回语句等基本结构。

## 3. ``$ gcc -o main main.s``

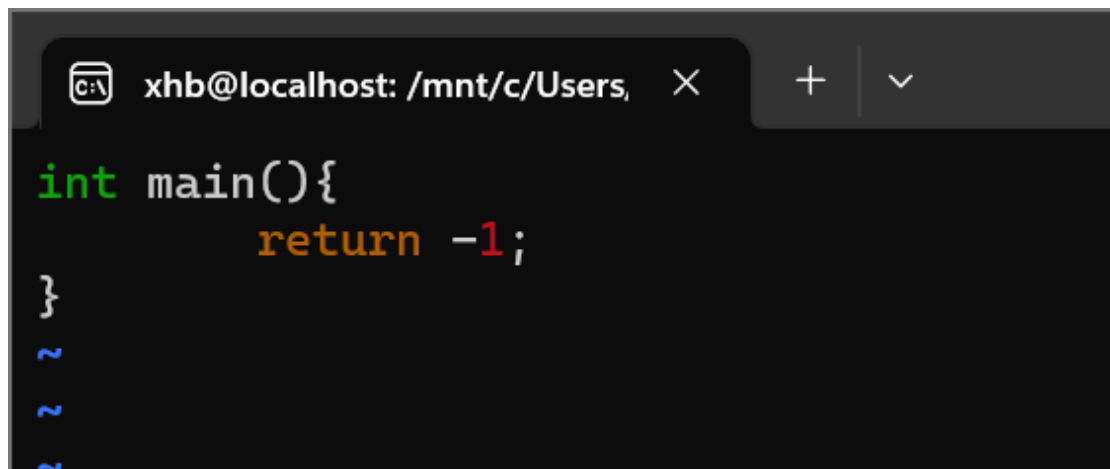
- 这个命令将汇编文件 ``main.s`` 编译并链接成可执行文件 ``main``。
- ``-o main`` 指定输出的可执行文件名为 ``main``。

这个命令本身不会在命令行上显示任何东西，但它会生成一个可执行文件 ``main``。

## 4. ``$ ./main``

- 这个命令运行名为 ``main`` 的可执行文件。
- 由于程序本身没有任何输出，且只是返回 `0`，执行这个命令后不会在命令行上看到任何输出。但你可以通过 ``echo $?`` 命令来查看上一个命令的退出状态，应该会显示 ``0``，表示程序成功执行并正常退出。

将 `main.c` 中的 `return 0;` 改为 `return -1;`；重复 1)、小题的所有的命令，给出命令结果，解释命令 `echo $?` 的输出结果及原因？



```
xhb@localhost: /mnt/c/Users, X + v
int main(){
    return -1;
}
~
~
~
```

```
xhb@localhost:/mnt/c/Users/洗胡兵$ vi main.c
xhb@localhost:/mnt/c/Users/洗胡兵$ gcc -g -S -o main.s main.c
xhb@localhost:/mnt/c/Users/洗胡兵$ cat main.s
        .file      "main.c"
        .text
.Ltext0:
        .file 0 "/mnt/c/Users/洗胡兵/" "main.c"
        .globl   main
        .type    main, @function
main:
.LFB0:
        .file 1 "main.c"
        .loc 1 1 11
        .cfi_startproc
        endbr64
        pushq    %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq     %rsp, %rbp
        .cfi_def_cfa_register 6
        .loc 1 2 9
        movl     $-1, %eax
        .loc 1 3 1
        popq     %rbp
        .cfi_def_cfa 7, 8
        ret
```



```
xhb@localhost: /mnt/c/Users/  × + ▾

    .byte    0
    .section      .debug_aranges,"",@progbits
    .long    0x2c
    .value   0x2
    .long    .Ldebug_info0
    .byte    0x8
    .byte    0
    .value   0
    .value   0
    .quad    .Ltext0
    .quad    .Ltext0-.Ltext0
    .quad    0
    .quad    0
    .section      .debug_line,"",@progbits
.Ldebug_line0:
    .section      .debug_str,"MS",@progbits,1
.LASF2:
    .string "GNU C17 11.4.0 -mtune=generic -march=x86-64 -g -fasynchronous-unwind-t
ables -fstack-protector-strong -fstack-clash-protection -fcf-protection"
.LASF3:
    .string "main"
    .section      .debug_line_str,"MS",@progbits,1
.LASF1:
    .string "/mnt/c/Users/\345\206\274\350\203\241\345\205\265"
.LASF0:
    .string "main.c"
    .ident "GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0"
    .section      .note.GNU-stack,"",@progbits
    .section      .note.gnu.property,"a"
    .align 8
    .long    1f - 0f
    .long    4f - 1f
    .long    5
0:
    .string "GNU"
1:
    .align 8
    .long    0xc0000002
    .long    3f - 2f
2:
    .long    0x3
3:
    .align 8
4:
xhb@localhost:/mnt/c/Users/洗胡兵$ gcc -o main main.s
xhb@localhost:/mnt/c/Users/洗胡兵$ ./main
xhb@localhost:/mnt/c/Users/洗胡兵$ echo $?
255
```

解释:

在大多数 Unix 和 Linux 系统中,进程的退出状态被定义为一个无符号的 8 位整数,这意味着返回值实际上是通过模 256 运算得到的,因此 `-1` 被转换为 `255`。

这种转换是因为在 Unix 和类 Unix 系统中,进程的退出状态(或返回代码)被存储为一个无符号的单字节整数。因此,负数被解释为其无符号等价物,这就是为什么返回 `-1` 实际上会导致退出状态为 `255` 的原因。

2)、将下述汇编代码保存为 `testmain.s`

```
.global main
```

```
main:
    movq $0, %rax
    ret
```

运行以下命令，给出命令结果，并解释命令的功能

```
$ gcc -o testmain testmain.s
$ ./testmain
$ echo $?
```

```
xhb@localhost:/mnt/c/Users/洗胡兵$ vi testmain.s
xhb@localhost:/mnt/c/Users/洗胡兵$ gcc -o testmain testmain.s
xhb@localhost:/mnt/c/Users/洗胡兵$ ./testmain
xhb@localhost:/mnt/c/Users/洗胡兵$ echo $?
0
```

解释：这段代码定义了一个名为 `main` 的全局标签（在 C 程序中对应于 `main` 函数）。`movq $0, %rax` 指令将立即数 `0` 移动到 `rax` 寄存器中，`rax` 寄存器在返回时用于存储函数的返回值（在 x86\_64 架构下）。`ret` 指令从函数返回

#### 1. `$ gcc -o testmain testmain.s`

这个命令使用 GCC 编译器将汇编源文件 `testmain.s` 编译并链接成名为 `testmain` 的可执行文件。GCC 在这个步骤中完成了汇编、链接等工作，生成了可以在操作系统上直接运行的程序。

#### 2. `$ ./testmain`

运行编译后的程序 `testmain`。由于程序不执行任何输出操作，不会在终端看到任何输出结果。

#### 3. `$ echo $?`

这条命令打印上一个命令（在这个例子中是 `./testmain`）的退出状态。根据 `testmain.s` 中的代码，我们知道程序的返回值应该是 `0`，因为那是我们显式设置的值。

3)、我们可以将上述 `movq $0, %rax` 改成其他数字如 `movq $-2, %rax`，重新编译测试结果。

```
xhb@localhost: /mnt/c/Users/洗胡兵$ vi testmain.s
.global main
main:
    movq $-2, %rax
    ret

xhb@localhost:/mnt/c/Users/洗胡兵$ vi testmain.s
xhb@localhost:/mnt/c/Users/洗胡兵$ gcc -o testmain testmain.s
xhb@localhost:/mnt/c/Users/洗胡兵$ ./testmain
xhb@localhost:/mnt/c/Users/洗胡兵$ echo $?
254
```

4)、将下面文件保存到 helloworld.s

```
.section .data

message:
    .ascii "hello world!\n"
    length = . - message

.section .text
.global main

main:
    movq $1, %rax
    movq $1, %rdi
    lea message(%rip), %rsi
    movq $length, %rdx
    syscall

    movq $60, %rax
    xor %rdi, %rdi
    syscall
```

运行以下命令，给出命令结果，并解释命令的功能

```
$ gcc -o helloworld helloworld.s
$ ./helloworld
$ echo $?
```

```
xhb@localhost:/mnt/c/Users/洗胡兵$ vi helloworld.s
xhb@localhost:/mnt/c/Users/洗胡兵$ gcc -o helloworld helloworld.s
xhb@localhost:/mnt/c/Users/洗胡兵$ ./helloworld
hello world!
xhb@localhost:/mnt/c/Users/洗胡兵$ echo $?
0
xhb@localhost:/mnt/c/Users/洗胡兵$
```

解释：

1. `$ gcc -o helloworld helloworld.s`

这个命令用 GCC 编译器将汇编源文件 `helloworld.s` 编译并链接成名为 `helloworld` 的可执行文件。没有错误的话，这个命令不会产生任何输出。

2. `$ ./helloworld`

运行编译后的程序。会在屏幕上看到输出 "hello world!"。

3. `$ echo $?`

这条命令打印上一个命令（即 `./helloworld`）的退出状态。由于程序使用 `xor %rdi, %rdi` 后执行 `exit` 系统调用来退出，因此期望的退出状态是 `0`。

5)、进程在内存中的位置安排

```
#include <stdio.h>

#include <stdlib.h>


int glob1=5;

int glob2=3;


char arr[10];

extern char **environ;


int add(int a,int b)

{
```

```
int sum;

sum = a+b;

printf(" sum: %p , value: %d\n",&sum,sum);

printf(" a: %p , value: %d\n",&a,a);

printf(" b: %p , value: %d\n\n",&b,b);

return(sum);

}


int main(int argc, char *argv[])

{

int sum;

char *p;

printf(" env: %p , value: %s\n\n",environ,*environ);

printf(" argc: %p , value: %d\n\n",&argc,argc);

printf(" sum: %p , value: %d\n\n",&sum,sum);

sum=add(glob1,glob2);


p=malloc(10);

*p = ' ';

printf(" p: %p , value: %d\n\n",p,*p);


printf(" arr: %p , value: %d\n",arr,arr[0]);

printf("glob2: %p , value: %d\n",&glob2,glob2);

printf("glob1: %p , value: %d\n\n",&glob1,glob1);
```

```

printf(" main: %p\n",main);

printf(" add: %p\n\n",add);

printf("sum=%d\n\n",sum);

return(sum);

}

```

将以上程序命名为 memory.c，在你的 linux 环境中运行以下命令，给出命令结果，并解释命令结果。

```

$ gcc -S memory.c
$ cat memory.s
$ gcc memory.c -o memory
$ ./memory
$ env
$ objdump -d memory

```

```

xhb@localhost:/mnt/c/Users/洗胡兵$ gcc -S memory.c
xhb@localhost:/mnt/c/Users/洗胡兵$ cat memory.s
        .file     "memory.c"
        .text
        .globl   glob1
        .data
        .align   4
        .type    glob1, @object
        .size    glob1, 4
glob1:
        .long    5
        .globl   glob2
        .align   4
        .type    glob2, @object
        .size    glob2, 4
glob2:
        .long    3
        .globl   arr
        .bss
        .align   8
        .type    arr, @object
        .size    arr, 10
arr:
        .zero    10
        .section      .rodata
.LC0:
        .string   "  sum: %p , value: %d\n"
.LC1:
        .string   "    a: %p , value: %d\n"
.LC2:
        .string   "    b: %p , value: %d\n\n"
        .text
        .globl   add
        .type    add, @function
add:
.LFB6:
        .cfi_startproc

```

```

xhb@localhost:/mnt/c/Users/洗胡兵$ gcc memory.c -o memory
xhb@localhost:/mnt/c/Users/洗胡兵$ ./memory
env: 0x7ffdc5894ba8 , value: SHELL=/bin/bash

argc: 0x7ffdc5894a5c , value: 1

sum: 0x7ffdc5894a6c , value: 0

sum: 0x7ffdc5894a34 , value: 8
a: 0x7ffdc5894a2c , value: 5
b: 0x7ffdc5894a28 , value: 3

p: 0x555bdcf26b0 , value: 32

arr: 0x555bdc274030 , value: 0
glob2: 0x555bdc274014 , value: 3
glob1: 0x555bdc274010 , value: 5

main: 0x555bdc271228
add: 0x555bdc271189

sum=8

```

```

xhb@localhost:/mnt/c/Users/洗胡兵$ env
SHELL=/bin/bash
WSL2_GUI_APPS_ENABLED=1
WSL_DISTRO_NAME=Ubuntu
NAME=localhost
PWD=/mnt/c/Users/洗胡兵
LOGNAME=xhb
MOTD_SHOWN=update-motd
HOME=/home/xhb
LANG=C.UTF-8
WSL_INTEROP=/run/WSL/359_interop
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.zst=01;31:*.tzt=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.wim=01;31:*.swm=01;31:*.dwm=01;31:*.esd=01;31:*.jpg=01;35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.webp=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=00;36:*.mid=00;36:*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
WAYLAND_DISPLAY=wayland-0
LESSCLOSE=/usr/bin/lesspipe %s %s
TERM=xterm-256color
LESSOPEN=| /usr/bin/lesspipe %s
USER=xhb
DISPLAY=:0
SHLVL=1
XDG_RUNTIME_DIR=/run/user/1000/
WSLENV=
XDG_DATA_DIRS=/usr/local/share:/usr/share:/var/lib/snapd/desktop

```

```
xhb@localhost:/mnt/c/Users/洗胡兵$ objdump -d memory
memory:      file format elf64-x86-64

Disassembly of section .init:

0000000000001000 <_init>:
 1000:      f3 0f 1e fa      endbr64
 1004:      48 83 ec 08      sub    $0x8,%rsp
 1008:      48 8b 05 d9 2f 00 00 mov    0x2fd9(%rip),%rax      # 3fe8 <__gmon_
start__@Base>
 100f:      48 85 c0          test   %rax,%rax
 1012:      74 02             je     1016 <_init+0x16>
 1014:      ff d0             call   *%rax
 1016:      48 83 c4 08      add    $0x8,%rsp
 101a:      c3               ret

Disassembly of section .plt:

0000000000001020 <.plt>:
 1020:      ff 35 8a 2f 00 00      push   0x2f8a(%rip)      # 3fb0 <_GLOBAL_OFFS
ET_TABLE_+0x8>
 1026:      f2 ff 25 8b 2f 00 00      bnd jmp *0x2f8b(%rip)      # 3fb8 <_GLOBAL_OF
FSET_TABLE_+0x10>
 102d:      0f 1f 00          nopl    (%rax)
 1030:      f3 0f 1e fa      endbr64
 1034:      68 00 00 00 00      push    $0x0
 1039:      f2 e9 e1 ff ff ff      bnd jmp 1020 <_init+0x20>
 103f:      90               nop
 1040:      f3 0f 1e fa      endbr64
 1044:      68 01 00 00 00      push    $0x1
 1049:      f2 e9 d1 ff ff ff      bnd jmp 1020 <_init+0x20>
 104f:      90               nop
```

解释:

(1) \$ gcc -S memory.c: 编译为汇编代码, 这个命令告诉 GCC 编译器将 C 源代码文件 `memory.c` 编译成汇编代码文件 `memory.s`。在这个步骤中, 编译器将 C 代码转换成汇编语言代码, 但不会进行进一步的编译或链接。

(2) \$ cat memory.s: 查看生成的汇编代码,

(3) \$ gcc memory.c -o memory: 编译为可执行文件

(4) \$ ./memory 运行可执行文件

(5) \$ env: 查看环境变量, 这个命令用于显示当前 **shell** 的环境变量。它输出了当前环境中设置的所有环境变量及其值。



(6) `$ objdump -d memory`: 查看可执行文件的汇编代码, 这个命令使用 `objdump` 工具查看可执行文件 `memory` 的反汇编结果, 也就是它的汇编代码。这将显示程序的机器码以及相应的汇编指令。

通过执行这些命令, 可以看到:

- 编译器生成的汇编代码 (`memory.s`);
- 编译器生成的可执行文件 (`memory`);
- 程序在运行时的输出, 包括变量地址、变量值以及一些其他信息;
- 当前 `shell` 环境中的环境变量;
- 可执行文件的汇编代码, 包括其机器码。

## 8、GDB 调测程序

<https://bbs.huaweicloud.com/blogs/308343>

### Linux GDB 调试基础

参照以上网页的步骤, 在你的电脑上, 重复网页中的步骤, 掌握 `gdb` 最基本的使用。

```
xhb@localhost:/mnt/c/Users/洗胡兵$ vi gdbtest.c
xhb@localhost:/mnt/c/Users/洗胡兵$ gcc -o gdbtest gdbtest.c -g
xhb@localhost:/mnt/c/Users/洗胡兵$ ./gdbtest
result=55
```

```
xhb@localhost:/mnt/c/Users/洗胡兵$ gdb gdbtest
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.htm
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 gdbtest...
(gdb)
```

```
xhb@localhost: /mnt/c/Users, X + v
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from gdbtest...
(gdb) start
Temporary breakpoint 1 at 0x1182: file gdbtest.c, line 12.
Starting program: /mnt/c/Users/洗胡兵/gdbtest
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Temporary breakpoint 1, main () at gdbtest.c:12
12      result = add(1, 10);
(gdb) step
add (start=1, end=10) at gdbtest.c:5
5      for(i=start; i<=end; i++)
(gdb) backtrace
#0  add (start=1, end=10) at gdbtest.c:5
#1  0x0000555555555191 in main () at gdbtest.c:12
(gdb) info locals
i = 395049983
sum = 0
(gdb) frame 1
#1  0x0000555555555191 in main () at gdbtest.c:12
12      result = add(1, 10);
(gdb) info locals
result = 21845
(gdb) set var sum=100
No symbol "sum" in current context.
(gdb) frame 0
#0  add (start=1, end=10) at gdbtest.c:5
5      for(i=start; i<=end; i++)
(gdb) print sum
$1 = 0
(gdb) frame 0
#0  add (start=1, end=10) at gdbtest.c:5
5      for(i=start; i<=end; i++)
(gdb) set var sum=100
(gdb) print sum
$2 = 100
(gdb) info locals
i = 395049983
sum = 100
(gdb) finish
Run till exit from #0  add (start=1, end=10) at gdbtest.c:5
0x0000555555555191 in main () at gdbtest.c:12
12      result = add(1, 10);
Value returned is $3 = 155
(gdb)
```

Wsl 环境下的 debug

<https://code.visualstudio.com/docs/cpp/config-wsl>

9、了解 fork 函数功能。（如何创建进程）

参考网页，

<https://zhuanlan.zhihu.com/p/53527981>

<https://www.cnblogs.com/cxuanBlog/p/13277369.html>

了解进程概念。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
```

```
int glob1=5;
```

```
int main(int argc, char *argv[])
{
```

```
int sum;
```

```
int ret_pid;
```

```
    ret_pid = fork(); // 第一个 fork
//  ret_pid = fork(); //第二个 fork
    printf("\n ppid=%d, pid=%d, ret_pid=%d \n",getppid(),getpid(),ret_pid);

    sleep(30); pause();
    return 0;
}
```

将以上程序命名为 fork1.c，在你的 linux 环境中运行以下命令，给出命令结果，并解释命令结果。（黄色部分要替换成自己的）

```
$ vi fork1.c
$ gcc fork1.c -o fork1
$ echo $$
15288
$ pstree 15288 -p
$ ./fork1 &
$ pstree 15288 -p
```

```

xhb@localhost:/mnt/c/Users/洗胡兵$ vi fork1.c
xhb@localhost:/mnt/c/Users/洗胡兵$ gcc fork1.c -o fork1
xhb@localhost:/mnt/c/Users/洗胡兵$ echo $$
360
xhb@localhost:/mnt/c/Users/洗胡兵$ pstree 360 -p
bash(360)——pstree(41280)
xhb@localhost:/mnt/c/Users/洗胡兵$ ./fork1 &
[1] 41353
xhb@localhost:/mnt/c/Users/洗胡兵$
  ppid=360, pid=41353, ret_pid=41354

  ppid=41353, pid=41354, ret_pid=0

xhb@localhost:/mnt/c/Users/洗胡兵$ pstree 360 -p
bash(360)——fork1(41353)——fork1(41354)
               |
               └─pstree(41462)
xhb@localhost:/mnt/c/Users/洗胡兵$

```

解释：

- 在编辑和编译之后，确定了当前 shell 进程的 PID 为 **360**。
- **pstree** 命令用于显示进程树。在运行程序之前和之后，我们使用它来观察进程树的变化。
- 在程序运行之前，只有一个进程（**360**，即当前 shell 进程）。
- 在程序运行后，由于程序中存在 **fork** 调用，进程树将会发生变化。**fork** 调用会创建一个新的子进程，该子进程会复制父进程的代码和数据，然后继续执行。因此，在执行 **./fork1 &** 后，会创建一个新的子进程，此时进程树中会出现两个具有相同代码和数据的进程，它们的 PID 将不同

将第 2 个 **fork()** 语句最前面的注解//取消（即第 2 个 **fork()** 起作用）。

运行以下命令，给出命令结果，并解释命令结果。

```
$ gcc fork1.c -o fork2
```

```
$ echo $$
```

```
15288
```

```
$ pstree 15288 -p
```

```
$ ./fork2 &
```

```
$ pstree 15288 -p
```

```

xhb@localhost:/mnt/c/Users/洗胡兵$ vi fork1.c
xhb@localhost:/mnt/c/Users/洗胡兵$ gcc fork1.c -o fork1
xhb@localhost:/mnt/c/Users/洗胡兵$ echo $$
360
xhb@localhost:/mnt/c/Users/洗胡兵$ pstree 360 -p
bash(360)─fork1(41353)─fork1(41354)
              └─pstree(42516)
xhb@localhost:/mnt/c/Users/洗胡兵$ ./fork1 &
[2] 42544
xhb@localhost:/mnt/c/Users/洗胡兵$

ppid=360, pid=42544, ret_pid=42547
ppid=42544, pid=42545, ret_pid=42546

ppid=42545, pid=42546, ret_pid=0
ppid=42544, pid=42547, ret_pid=0

xhb@localhost:/mnt/c/Users/洗胡兵$ pstree 360 -p
bash(360)─fork1(41353)─fork1(41354)
              └─fork1(42544)─fork1(42545)─fork1(42546)
                  └─fork1(42547)
                    └─pstree(42580)
xhb@localhost:/mnt/c/Users/洗胡兵$

```

解释：

- 编译程序后，我们确定当前 shell 进程的 PID 是 **360**。
- 在运行程序之前，通过 **pstree** 命令查看了进程树。此时只有一个进程，即当前 shell 进程。
- 在程序运行后，由于第二个 **fork()** 调用未被注释，因此程序将创建第二个子进程。这样，进程树中将有四个进程：当前 shell 进程、第一个子进程、第二个子进程以及他们的孙子进程。
- 第一个子进程是由第一个 **fork()** 调用创建的，而第二个子进程是由第二个 **fork()** 调用创建的。
- 孙子进程是由第二个子进程的 **fork()** 调用创建的。
- 因此，在运行 **./fork2 &** 之后，进程树中会出现四个进程，而不是两个

10、了解 fork 的二次返回值。（如何区分父、子进程）

```

#include <unistd.h>
#include <stdio.h>
int main ()
{
    pid_t fpid; //fpid 表示 fork 函数返回的值

```

```

int count=0;
fpid=fork();
if (fpid < 0)
    printf("error in fork!");
else if (fpid == 0) {
    printf(" fpid=%d,pid=%d,ppid=%d\n",
        fpid,getppid(),getpid());
    count+=10;
}
else {
    printf(" fpid=%d,pid=%d,ppid=%d\n",
        fpid,getppid(),getpid());
    count+=20;
}
printf(" fpid=%d, count=%d\n",fpid,count);
sleep(60); pause();
}

```

将以上程序命名为 fork3.c，在你的 linux 环境中运行以下命令，给出命令结果，并解释命令结果。

(黄色部分要替换成自己的)

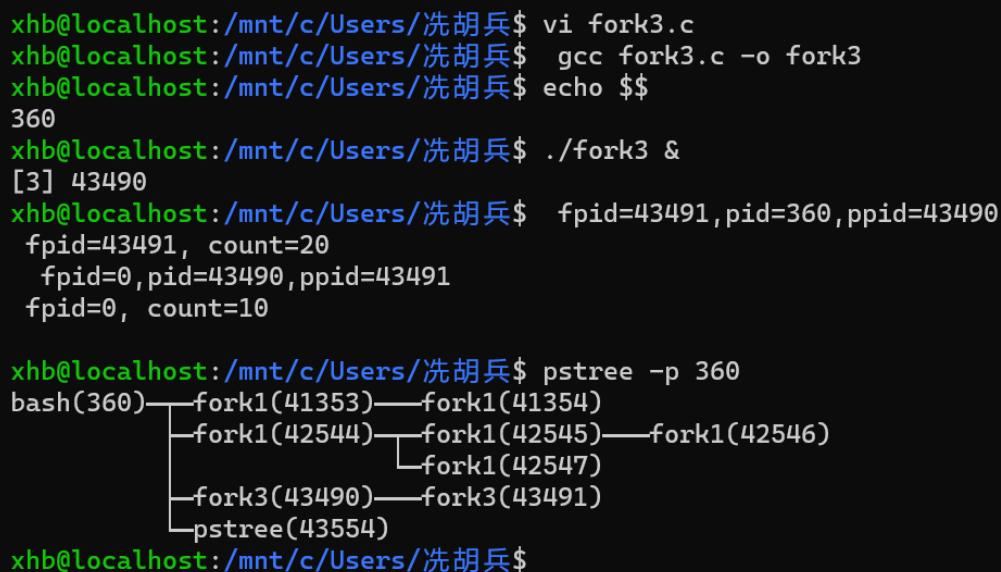
```
$ gcc fork3.c -o fork3
```

```
$ echo $$
```

```
16314
```

```
$ ./fork3 &
```

```
$ pstree -p 16314
```



```

xhb@localhost:/mnt/c/Users/洗胡兵$ vi fork3.c
xhb@localhost:/mnt/c/Users/洗胡兵$ gcc fork3.c -o fork3
xhb@localhost:/mnt/c/Users/洗胡兵$ echo $$
360
xhb@localhost:/mnt/c/Users/洗胡兵$ ./fork3 &
[3] 43490
xhb@localhost:/mnt/c/Users/洗胡兵$ fpid=43491,pid=360,ppid=43490
fpid=43491, count=20
fpid=0,pid=43490,ppid=43491
fpid=0, count=10

xhb@localhost:/mnt/c/Users/洗胡兵$ pstree -p 360
bash(360)─fork1(41353)─fork1(41354)
          │
          └─fork1(42544)─fork1(42545)─fork1(42546)
                    │
                    └─fork1(42547)
          └─fork3(43490)─fork3(43491)
                └─pstree(43554)
xhb@localhost:/mnt/c/Users/洗胡兵$

```

解释：

- 在编译程序后，我们确定当前 shell 进程的 PID 是 16314。

- 在程序运行后，会执行 `fork()` 函数，创建一个子进程。子进程将复制父进程的代码和数据，并在 `if` 或 `else` 语句中执行不同的代码。
- 在 `if` 语句中，子进程输出自己的 PID、父进程的 PID 和自身的 PID，并将 `count` 值增加 10。
- 在 `else` 语句中，父进程输出自己的 PID、子进程的 PID 和自身的 PID，并将 `count` 值增加 20。
- 无论是子进程还是父进程，最后都会输出 `fpid` 和 `count` 的值。
- 在程序的最后，使用 `sleep(60)` 和 `pause()` 函数使得程序暂停执行，这样可以保持进程在后台运行。

## 11、了解孤儿进程及被#1 进程收养。

（父进程先于子进程终止时，子进程的父进程处理方式）

用上一题目的程序。

在你的 linux 环境中运行以下命令，给出命令结果，并解释命令结果。（黄色部分要替换成自己的）

```
$ ps ax -o user,group,comm,ppid,pid,stat,ttty
$ echo $$
17058
$ pstree -p 17058
bash(17058)——pstree(17093)
$ gcc fork3.c -o fork5
$ ./fork5 &
$ pstree -p 17058
bash(17058)——fork5(17094)——fork5(17095)
└──pstree(17096)
$ kill 17094
$ pstree -p 17058
bash(17058)——pstree(17097)
[1]+ Terminated ./fork5
$ ps ax -o user,group,comm,ppid,pid,stat,ttty
USER  GROUP  COMMAND      PPID  PID  STAT TT
root  root   agetty        1    666  Ss+  ttyS0
root  root   agetty        1    677  Ss+  tty1
wang  wang   bash          16875 16912 Ss  pts/0
root  wang   su            16912 16941 S  pts/0
wang  wang   bash          16941 16942 S+  pts/0
wang  wang   bash          17021 17058 Ss  pts/1
wang  wang   fork5         1 17095 S  pts/1
wang  wang   ps            17058 17098 R+  pts/1
[wang@i223mw70pqbz ~/c 03.29 16:03:50 33] $
```

USER	GROUP	COMMAND	PPID	PID	STAT	TT
root	root	systemd	0	1	Ss	?
root	root	init-systemd(Ub	1	2	SL	?
root	root	init	2	5	SL	?
root	root	systemd-journal	1	34	S<s	?
root	root	systemd-udev	1	53	Ss	?
root	root	snappy	1	65	Ss	?
root	root	snappy	1	66	Ss	?
root	root	snappy	1	67	Ss	?
root	root	snappy	1	81	Ss	?
root	root	snappy	1	82	Ss	?
root	root	snappy	1	84	Ss	?
root	root	snappy	1	88	Ss	?
root	root	snappy	1	90	Ss	?
systemd+	systemd+	systemd-resolve	1	120	Ss	?
root	root	cron	1	137	Ss	?
message+	message+	dbus-daemon	1	140	Ss	?
root	root	networkd-dispat	1	154	Ss	?
syslog	syslog	rsyslogd	1	157	Ssl	?
root	root	snappy	1	161	Ssl	?
root	root	systemd-logind	1	176	Ss	?
root	root	subiquity-serve	1	205	Ss	?
root	root	unattended-upgr	1	209	Ssl	?
root	root	agetty	1	212	Ss+	hvc0
root	root	agetty	1	219	Ss+	tty1
root	root	sshd	1	220	Ss	?
root	root	python3.10	205	330	SL	?
root	root	SessionLeader	2	358	Ss	?
root	root	Relay(360)	358	359	R	?
xhb	xhb	bash	359	360	Ss	pts/0
root	xhb	login	2	361	Ss	pts/1
xhb	xhb	systemd	1	405	Ss	?
xhb	xhb	(sd-pam)	405	412	S	?
xhb	xhb	bash	361	425	S+	pts/1
root	root	python3	330	444	S	?
root	root	packagekitd	1	39150	Ssl	?
root	root	polkitd	1	39154	Ssl	?
xhb	xhb	fork1	360	41353	S	pts/0
xhb	xhb	fork1	41353	41354	S	pts/0
xhb	xhb	fork1	360	42544	S	pts/0
xhb	xhb	fork1	42544	42545	S	pts/0
xhb	xhb	fork1	42545	42546	S	pts/0
xhb	xhb	fork1	42544	42547	S	pts/0
xhb	xhb	fork3	360	43490	S	pts/0
xhb	xhb	fork3	43490	43491	S	pts/0
xhb	xhb	ps	360	44067	R+	pts/0



```

xhb@localhost:/mnt/c/Users/洗胡兵$ echo $$
360
xhb@localhost:/mnt/c/Users/洗胡兵$ pstree -p 360
bash(360)─fork1(41353)─fork1(41354)
          │
          └─fork1(42544)─fork1(42545)─fork1(42546)
                    │
                    └─fork1(42547)
          └─fork3(43490)─fork3(43491)
                └─pstree(44232)

xhb@localhost:/mnt/c/Users/洗胡兵$ gcc fork3.c -o fork5
xhb@localhost:/mnt/c/Users/洗胡兵$ ./fork5 &
[4] 44534
xhb@localhost:/mnt/c/Users/洗胡兵$  fpid=44535,pid=360,ppid=44534
    fpid=44535, count=20
    fpid=0,pid=44534,ppid=44535
    fpid=0, count=10

xhb@localhost:/mnt/c/Users/洗胡兵$ pstree -p 360
bash(360)─fork1(41353)─fork1(41354)
          │
          └─fork1(42544)─fork1(42545)─fork1(42546)
                    │
                    └─fork1(42547)
          └─fork3(43490)─fork3(43491)
                └─fork5(44534)─fork5(44535)
                        └─pstree(44686)

xhb@localhost:/mnt/c/Users/洗胡兵$ kill 42544
xhb@localhost:/mnt/c/Users/洗胡兵$ pstree -p 360
bash(360)─fork1(41353)─fork1(41354)
          │
          └─fork3(43490)─fork3(43491)
                └─fork5(44534)─fork5(44535)
                        └─pstree(44810)

[2] Terminated ./fork1

```

```
xhb@localhost: /mnt/c/Users/冼胡兵$ ps ax -o user,group,comm,ppid,pid,stat,TTY
USER      GROUP    COMMAND                PPID    PID  STAT  TT
root      root     systemd                0        1  Ss    ?
root      root     init-systemd(Ub        1        2  Sl    ?
root      root     init                   2        5  Sl    ?
root      root     systemd-journal       1       34  S<s   ?
root      root     systemd-udev          1       53  Ss    ?
root      root     snapfuse              1       65  Ss    ?
root      root     snapfuse              1       66  Ss    ?
root      root     snapfuse              1       67  Ss    ?
root      root     snapfuse              1       81  Ss    ?
root      root     snapfuse              1       82  Ss    ?
root      root     snapfuse              1       84  Ss    ?
root      root     snapfuse              1       88  Ss    ?
root      root     snapfuse              1       90  Ss    ?
systemd+  systemd+ systemd-resolve        1      120  Ss    ?
root      root     cron                   1      137  Ss    ?
message+  message+ dbus-daemon            1      140  Ss    ?
root      root     networkd-dispat       1      154  Ss    ?
syslog    syslog   rsyslogd              1      157  Ssl   ?
root      root     snapd                  1      161  Ssl   ?
root      root     systemd-logind        1      176  Ss    ?
root      root     subiquity-serve       1      205  Ss    ?
root      root     unattended-upgr       1      209  Ssl   ?
root      root     agetty                 1      212  Ss+   hvc0
root      root     agetty                 1      219  Ss+   tty1
root      root     sshd                   1      220  Ss    ?
root      root     python3.10            205     330  Sl    ?
root      root     SessionLeader         2      358  Ss    ?
root      root     Relay(360)            358     359  R     ?
xhb       xhb      bash                   359     360  Ss    pts/0
root      xhb      login                  2      361  Ss    pts/1
xhb       xhb      systemd               1      405  Ss    ?
xhb       xhb      (sd-pam)              405     412  S     ?
xhb       xhb      bash                   361     425  S+    pts/1
root      root     python3                330     444  S     ?
root      root     packagekitd           1     39150  Ssl   ?
root      root     polkitd                1     39154  Ssl   ?
xhb       xhb      fork1                  360    41353  S     pts/0
xhb       xhb      fork1                  41353  41354  S     pts/0
xhb       xhb      fork1                  359    42545  S     pts/0
xhb       xhb      fork1                  42545  42546  S     pts/0
xhb       xhb      fork1                  359    42547  S     pts/0
xhb       xhb      fork3                  360    43490  S     pts/0
xhb       xhb      fork3                  43490  43491  S     pts/0
xhb       xhb      fork5                  360    44534  S     pts/0
xhb       xhb      fork5                  44534  44535  S     pts/0
xhb       xhb      ps                     360    44928  R+    pts/0
xhb@localhost: /mnt/c/Users/冼胡兵$
```

12、进程打开文件的权限测试。

（进程对文件操作权限）

```
#include <stdio.h>
```

```
#include <string.h>
```

```
#include <stdlib.h>
```

```
#include <unistd.h>
```

```
int main(int argc, char* argv[])
```

```
{
```

```
    FILE *fp;
```

```

int ch,count=0;
printf("pid:%d, ppid:%d, euid:%d, egid:%d\n", getpid(), getppid(), geteuid(), getegid());
if (argc<2) { printf("usage:mycat file \n"); exit(-1);}
if((fp=fopen(argv[1], "r"))==NULL)
{
    printf("Can't open file %s\n",argv[1]);
    perror(argv[1]); // error information
    exit(1);
}
while( (ch=fgetc(fp)) != EOF)
{ printf("%02x ",ch);
    if(++count % 16 == 0) printf("\n");
}
printf("\n");
fclose(fp);
return(0);
}

```

在你的 linux 环境中运行以下命令，给出命令结果，并解释命令结果。

```

cd ~
vi testfile.c
gcc testfile.c -o testfile
echo $$
echo hello file > hello.txt
ls -l hello.txt testfile
# 以下命令中， testfile 为进程， hello.txt 为文件，测试进程对 hello.txt 文件的读权限
# $USER 为当前用户（按要求，必须前面创建的学号用户登入系统，运行以下命令）
./testfile hello.txt
chmod 000 hello.txt
ls -l hello.txt testfile
./testfile hello.txt
sudo ./testfile hello.txt
chmod 400 hello.txt
ls -l hello.txt testfile
./testfile hello.txt
chmod 040 hello.txt
ls -l hello.txt testfile
./testfile hello.txt
sudo chown root hello.txt
ls -l hello.txt testfile
./testfile hello.txt
sudo chmod 004 hello.txt
ls -l hello.txt testfile

```

```
./testfile hello.txt  
sudo chgrp root hello.txt  
ls -l hello.txt testfile  
./testfile hello.txt  
sudo chown $USER hello.txt  
ls -l hello.txt testfile  
./testfile hello.txt  
rm -r hello.txt
```

```

xhb@localhost:~$ chmod 400 hello.txt
xhb@localhost:~$ ls -l hello.txt testfile
-r----- 1 xhb xhb 11 Mar 23 17:54 hello.txt
-rwxr-xr-x 1 xhb xhb 16440 Mar 23 17:54 testfile
xhb@localhost:~$ ./testfile hello.txt
pid:50004, ppid:360, euid:1000, egid:1000
68 65 6c 6c 6f 20 66 69 6c 65 0a
xhb@localhost:~$ chmod 040 hello.txt
xhb@localhost:~$ ls -l hello.txt testfile
----r----- 1 xhb xhb 11 Mar 23 17:54 hello.txt
-rwxr-xr-x 1 xhb xhb 16440 Mar 23 17:54 testfile
xhb@localhost:~$ ./testfile hello.txt
pid:50112, ppid:360, euid:1000, egid:1000
Can't open file hello.txt
hello.txt: Permission denied
xhb@localhost:~$ sudo chown root hello.txt
xhb@localhost:~$ ls -l hello.txt testfile
----r----- 1 root xhb 11 Mar 23 17:54 hello.txt
-rwxr-xr-x 1 xhb xhb 16440 Mar 23 17:54 testfile
xhb@localhost:~$ ./testfile hello.txt
pid:50255, ppid:360, euid:1000, egid:1000
68 65 6c 6c 6f 20 66 69 6c 65 0a
xhb@localhost:~$ sudo chmod 004 hello.txt
xhb@localhost:~$ ls -l hello.txt testfile
-----r-- 1 root xhb 11 Mar 23 17:54 hello.txt
-rwxr-xr-x 1 xhb xhb 16440 Mar 23 17:54 testfile
xhb@localhost:~$ ./testfile hello.txt
pid:50355, ppid:360, euid:1000, egid:1000
Can't open file hello.txt
hello.txt: Permission denied
xhb@localhost:~$ sudo chgrp root hello.txt
xhb@localhost:~$ ls -l hello.txt testfile
-----r-- 1 root root 11 Mar 23 17:54 hello.txt
-rwxr-xr-x 1 xhb xhb 16440 Mar 23 17:54 testfile
xhb@localhost:~$ ./testfile hello.txt
pid:50486, ppid:360, euid:1000, egid:1000
68 65 6c 6c 6f 20 66 69 6c 65 0a
xhb@localhost:~$ sudo chown $xhb hello.txt
chown: missing operand after 'hello.txt'
Try 'chown --help' for more information.
xhb@localhost:~$ ls -l hello.txt testfile
-----r-- 1 root root 11 Mar 23 17:54 hello.txt
-rwxr-xr-x 1 xhb xhb 16440 Mar 23 17:54 testfile
xhb@localhost:~$ ./testfile hello.txt
pid:50613, ppid:360, euid:1000, egid:1000
68 65 6c 6c 6f 20 66 69 6c 65 0a
xhb@localhost:~$ rm -r hello.txt
rm: remove write-protected regular file 'hello.txt'? y
xhb@localhost:~$

```

解释:

上述命令创建了一个简单的 C 程序 `testfile.c`，用于读取文件内容。然后我们对 `hello.txt` 文件进行了不同权限的设置，并运行了 `testfile` 程序，观察了程序对文件的访问情况。

#### 1. 初始权限设置：

- `hello.txt` 文件的权限为 `-rw-r--r--`，即所有者可读写，组成员和其他用户只读。
- `testfile` 可执行文件权限为 `-rwxr-xr-x`，即所有者可读、写、执行，组成员和其他用户可执行。

#### 2. 测试程序对文件的读权限：

- 当 `hello.txt` 具有读权限时，程序可以成功打开文件并读取其内容。

#### 3. 修改文件权限：

- 使用 `chmod 000 hello.txt` 将文件权限设置为不允许任何人进行任何操作。
- 文件权限设置为 `-----`，即没有任何权限。

#### 4. 运行程序：

- 由于 `testfile` 没有权限访问 `hello.txt`，程序无法打开文件并读取其内容，输出错误信息。

#### 5. 使用 `sudo` 运行程序：

- 即使使用 `sudo` 运行程序，也无法打开文件，因为 `sudo` 不会改变文件的权限。

#### 6. 修改文件权限并再次运行：

- 将 `hello.txt` 的权限设置为只允许所有者读取后，程序可以成功读取文件内容。

#### 7. 再次修改文件权限并运行：

- 将 `hello.txt` 的权限设置为只允许组成员读取后，程序再次无法读取文件内容。

#### 8. 更改文件所有者并运行：

- 将 `hello.txt` 的所有者更改为 `root` 后，程序无法读取文件内容。

#### 9. 更改文件权限并运行：

- 将 `hello.txt` 的权限设置为只允许其他用户读取后，程序可以成功读取文件内容。

13、接上一题。（设置用户 ID 位程序）

（设置用户 ID 位如何起作用）

在你的 linux 环境中运行以下命令，给出命令结果，并解释命令结果。

```
cd ~
echo $$
echo hello file > hello.txt
chmod 600 hello.txt
ls -l hello.txt testfile
./testfile hello.txt
sudo chown root hello.txt
ls -l hello.txt testfile
./testfile hello.txt
sudo ./testfile hello.txt
sudo chown root testfile
ls -l testfile hello.txt
./testfile hello.txt
sudo chmod u+s testfile
ls -l testfile hello.txt
./testfile hello.txt
```

```

xhb@localhost:~$ cd ~
xhb@localhost:~$ echo $$
360
xhb@localhost:~$ echo hello file > hello.txt
xhb@localhost:~$ chmod 600 hello.txt
xhb@localhost:~$ ls -l hello.txt testfile
-rw----- 1 xhb xhb 11 Mar 23 18:20 hello.txt
-rwxr-xr-x 1 xhb xhb 16440 Mar 23 17:54 testfile
xhb@localhost:~$ ./testfile hello.txt
pid:52263, ppid:360, euid:1000, egid:1000
68 65 6c 6c 6f 20 66 69 6c 65 0a
xhb@localhost:~$ sudo chown root hello.txt
xhb@localhost:~$ ls -l hello.txt testfile
-rw----- 1 root xhb 11 Mar 23 18:20 hello.txt
-rwxr-xr-x 1 xhb xhb 16440 Mar 23 17:54 testfile
xhb@localhost:~$ ./testfile hello.txt
pid:52350, ppid:360, euid:1000, egid:1000
Can't open file hello.txt
hello.txt: Permission denied
xhb@localhost:~$ sudo ./testfile hello.txt
pid:52376, ppid:52375, euid:0, egid:0
68 65 6c 6c 6f 20 66 69 6c 65 0a
xhb@localhost:~$ sudo chown root testfile
xhb@localhost:~$ ls -l testfile hello.txt
-rw----- 1 root xhb 11 Mar 23 18:20 hello.txt
-rwxr-xr-x 1 root xhb 16440 Mar 23 17:54 testfile
xhb@localhost:~$ ./testfile hello.txt
pid:52440, ppid:360, euid:1000, egid:1000
Can't open file hello.txt
hello.txt: Permission denied
xhb@localhost:~$ sudo chmod u+s testfile
xhb@localhost:~$ ^C
xhb@localhost:~$ ls -l testfile hello.txt
-rw----- 1 root xhb 11 Mar 23 18:20 hello.txt
-rwsr-xr-x 1 root xhb 16440 Mar 23 17:54 testfile
xhb@localhost:~$
xhb@localhost:~$ ./testfile hello.txt
pid:52545, ppid:360, euid:0, egid:1000
68 65 6c 6c 6f 20 66 69 6c 65 0a
xhb@localhost:~$

```

解释:

### 1. 初始设置:

- 创建一个名为 `hello.txt` 的文件，内容为 "hello file"。
- 将 `hello.txt` 的权限设置为 `-rw-----`，即只有所有者有读写权限。
- `testfile` 可执行文件的权限为 `-rwxr-xr-x`。

### 2. 运行程序:



- 在没有设置用户 ID 位的情况下运行 `testfile` 程序，由于文件 `hello.txt` 的权限不允许其他用户读取，程序无法打开文件并读取内容。

### 3. 更改文件所有者：

- 将 `hello.txt` 的所有者更改为 `root`，但即使文件所有者为 `root`，程序依然无法打开文件。

### 4. 使用 `sudo` 运行程序：

- 使用 `sudo` 权限运行程序，程序依然无法打开文件。

### 5. 更改程序所有者：

- 将 `testfile` 的所有者更改为 `root`，但即使程序的所有者为 `root`，程序依然无法打开文件。

### 6. 设置用户 ID 位：

- 使用 `sudo chmod u+s testfile` 命令为 `testfile` 设置用户 ID 位。
- 查看文件权限，可以看到 `testfile` 的权限由 `-rwxr-xr-x` 变为 `-rwsr-xr-x`。

### 7. 再次运行程序：

- 此时，即使是普通用户执行 `testfile`，由于程序具有用户 ID 位，它以文件所有者的身份执行，因此可以成功打开并读取 `hello.txt` 文件的内容。