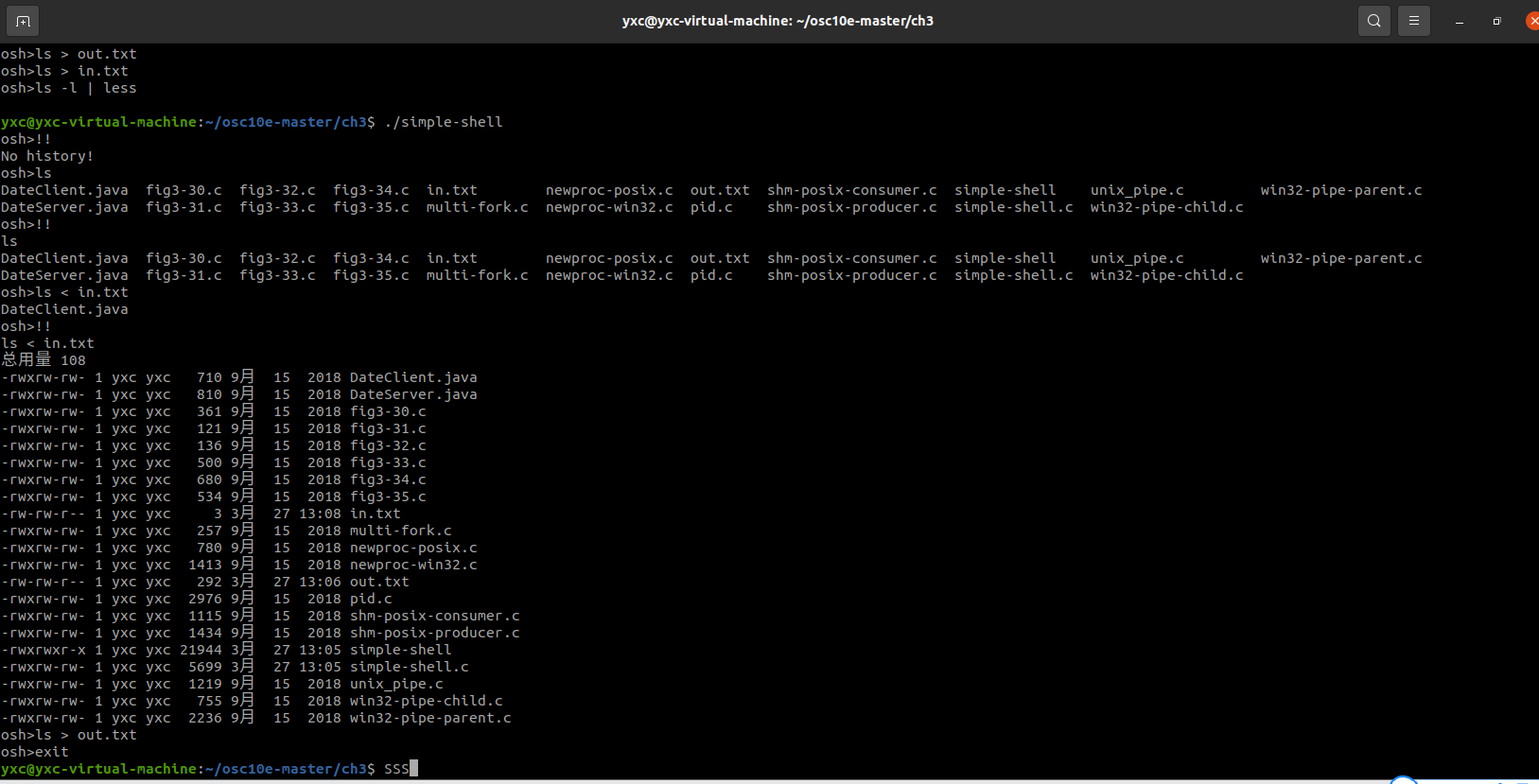
一：UNIX Shell

1. 首先将输入的字符串指令转化成execvp（）函数能接受的形式，存放在args字符串数组里。具体规则即是遇到空格则跳过，并将下一个字符连接在args数组中的下一个字符串元素末端。在实际实现中，我们用init\_args() 函数完此功能。
2. 此外，为了为用户提供历史记录功能，我们需要将当前的指令复制到到另一个字符串数组history中，需要注意的是，由于指令！！是调用历史指令，故若当前指令为！！，则不会对history进行修改，即保持更上一条历史记录。此功能用Init\_history()函数实现。
3. 自然，在将指令转化为字符串数组之后，将开始解析其意义，首先解决特殊指令（将改变当前指令或进程的指令）的操作。对于！！指令，需要的操作即是将history中记录的历史指令复制到当前指令数组args中并且将历史指令输出到窗口中。对于exit指令，则退出unix shell指令操作模式，即将should\_run置零。
4. 其次需要判断当前指令是否使用管道pipe，使用find\_pipe函数对当前指令进行检索，若存在“|“字符则就是存在pipe。
5. 若不存在pipe则是创建子进程，在子进程中，首先对指令进行输入和输出的重定向判定，用redirect\_input和redirect\_output两个函数对指令进行重定向。最后再执行指令。若指令末端不是“&”，则父进程需要等待，否则并行。
6. 若存在pipe则也创建子进程，在子进程中，再创建一个子进程，在子进程中仅执行第一条命令，并将输出通过管道重定向到第二条命令的输入；在子子进程中，从管道接受第一条指令的输出作为输入，然后执行第二条指令。父进程等待子进程的执行。

结果如图：

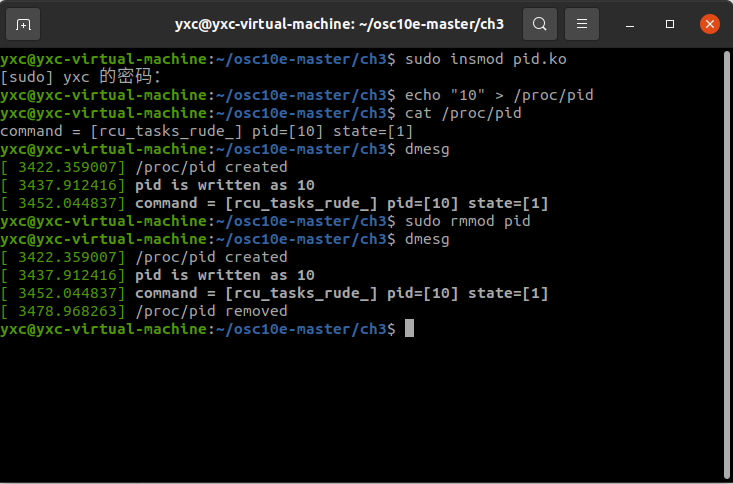
二、Linux Kernel Module for Task Information

1、根据要求修改源代码中的pid.c文件。需要注意的是，由于操作系统的版本差别，在我使用的系统中file\_operations数据类型并不适用，需要改为proc\_ops，并且其中的元素应改为.proc\_read和.proc\_write，分别与proc\_read()和proc\_write()两个函数对应。

2、在proc\_write()函数中，与书上原理部分所述不同的是，由于注释中写到“kstrol() will not work because the strings are not guaranteed to be null-terminated. sscanf() must be used instead.”即是kstrol()函数在字符串以空值结尾时不起作用，需要改为sscanf()函数将k\_mem中的字符串转化为l\_pid中的整型类型数。

3、在proc\_read()中，如原理部分所述，采用pid\_task()函数调用task\_struct结构体，若返回值为NULL，及代表pid无效，输出“invalid pid！“；若返回值不是NULL，则提取结构体中的comm和state元素，与l\_pid一起按照要求输出到用户界面上

4、需要注意的是，为了编译pid.c文件，我们借用了上个project中的makefile文件，在命令行中输入make指令，即可得到pid.ko内核模块。

结果如图