

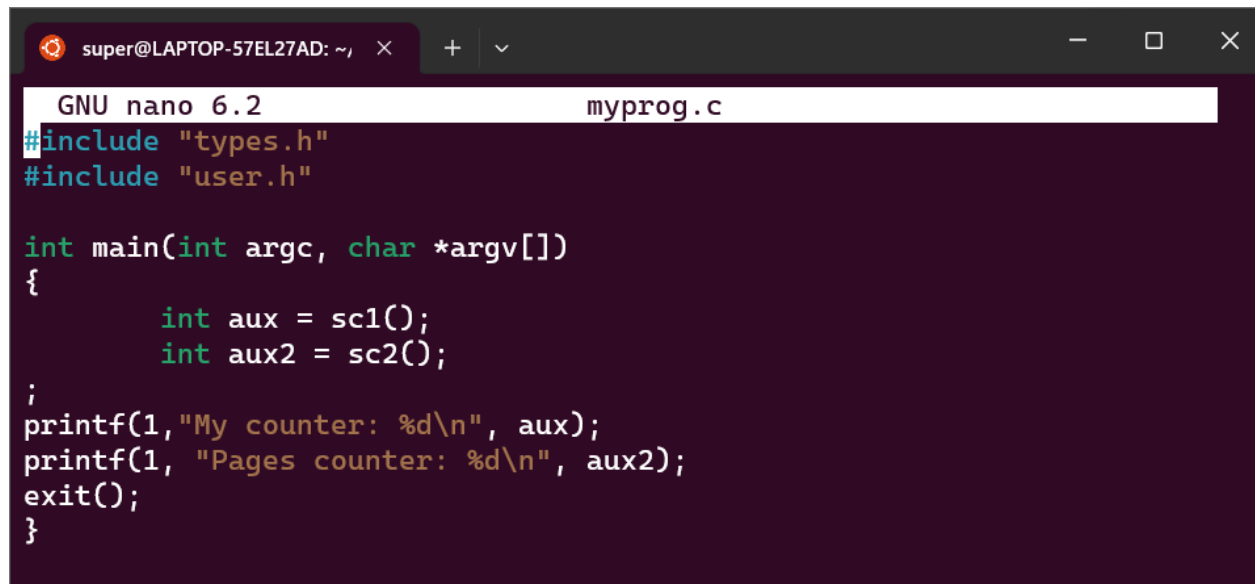
Questions 1 & 2.

Screenshots at the end of the document.

Question 3.

1. What files did you need to change and why did you need to change them?
 - a. `syscall.c`
 - i. Here we add the implementations of our custom-made system calls. Here is where we add our counter, getters and setters etc. We could say this is the main point where we want to create a system call.
 - b. `syscall.h`
 - i. We need to create a header for our new system calls in the array of system calls and assign the proper value to identify it afterwards.
 - c. `sysproc.c`
 - i. Here we add the functionality or the main code of the functions that we are working with. Here is where we write the code that operates our system calls. In other words, this is where we write what does or system calls do.
 - d. `usys.S`
 - i. I think this file is invoked by the assembly interruptions to each system call. When we interrupt, we trigger the system calls, and this file is responsible for this.
 - e. `user.h`
 - i. Here we just declare the system call functions that our user will be using.
 - f. `defs.h`
 - i. Here we store all the definitions. We are talking about variables, constants, data structures, functions, and function prototypes.
 - g. `myprog.c`
 - i. This is our program and we wanted to change it because we no longer wanted to print a "hello world", we wanted to show the number of times our system calls happened.
2. When your program calls the two system calls, what is occurring?
 - a. It first runs one of the two system calls. It goes to the kernel mode and will be running it until it finishes or when the interrupt comes along by some error. Then it will go for the next system call and will get the information when the interrupt hits. Between the two system calls the control does not come back to the user, it goes to the System Call handler to define what would we run next. I think it is trying to complete the user request handling the resources as it was taught, the best way possible.
3. Why did we need to edit or not edit the Makefile?
 - a. I did not edit the Makefile and I think we did not need to edit it because we already did that for our program. Now, we are calling the system calls from within the program, so the Makefile is still calling the program and the content of the program will be mostly irrelevant to the system call if it compiles with the rest of the xv6. We would need to edit it

we the files mentioned above in (a) were correctly linked but since we could run assignment 1, the work normally.



The image shows a terminal window with a dark background. At the top, a window title bar displays 'super@LAPTOP-57EL27AD: ~/ ' followed by standard window controls. Below this, the terminal header shows 'GNU nano 6.2' on the left and 'myprog.c' on the right. The main area contains C code with syntax highlighting: preprocessor directives are in blue, keywords in green, and identifiers/strings in yellow. The code defines a main function that calls two subroutines, prints their return values, and then exits.

```
GNU nano 6.2 myprog.c
#include "types.h"
#include "user.h"

int main(int argc, char *argv[])
{
    int aux = sc1();
    int aux2 = sc2();
;
printf(1, "My counter: %d\n", aux);
printf(1, "Pages counter: %d\n", aux2);
exit();
}
```

```
super@LAPTOP-57EL27AD: ~/  × + v
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ myprog
My counter: 7
Pages counter: 56790
$ myprog
exec: fail
exec myprog failed
$ myprog
My counter: 27
Pages counter: 113580
$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat        2 3 15532
echo       2 4 14412
forktest   2 5 8856
grep        2 6 18376
init        2 7 15032
kill        2 8 14500
ln          2 9 14396
ls          2 10 16964
mkdir       2 11 14520
rm          2 12 14500
sh          2 13 28556
stressfs    2 14 15432
usertests   2 15 62932
wc          2 16 15956
zombie      2 17 14080
myprog      2 18 14476
console     3 19 0
$ myprog
My counter: 70
Pages counter: 170370
$ █
```