**Lab #1: System Call Implementation**

Submitted by: Priyanka Jadli (862325015) and Chirag Rajavat (862324954)

# **Demonstration Link**

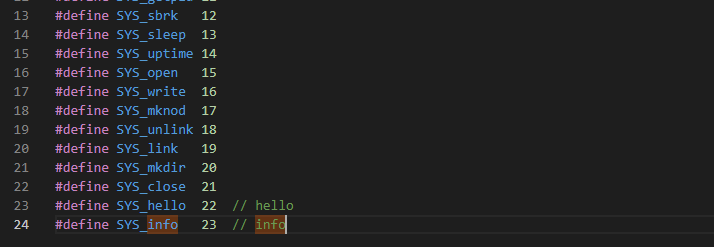
<https://youtu.be/Rvi1p6DWAkc>

# **List of files modified:**

1. kernel/sycall.h
2. kernel/syscall.c
3. kernel/sysproc.c
4. kernel/proc.h
5. kernel/proc.c
6. kernel/defs.h
7. user/usys.pl
8. user/user.h
9. Makefile
10. user/test.c

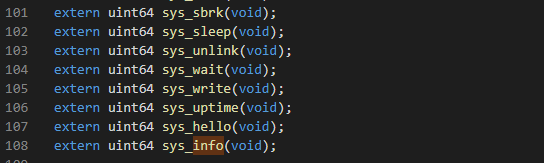
# **Modifications Screenshots**

## Syscall.h

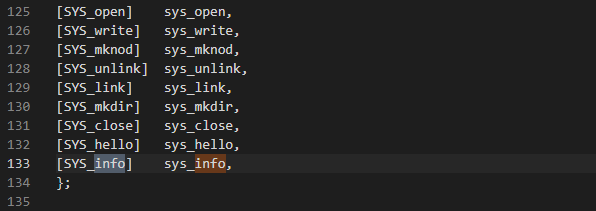
Assign a new number to our system call.

## Syscall.c

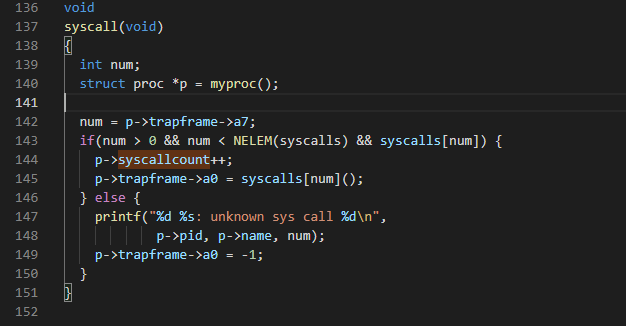
Added declaration for sys\_info syscall.



Added a pointer to the system call in the array. This array contains function pointers which uses numbers defined in syscall.h as pointers to system calls.

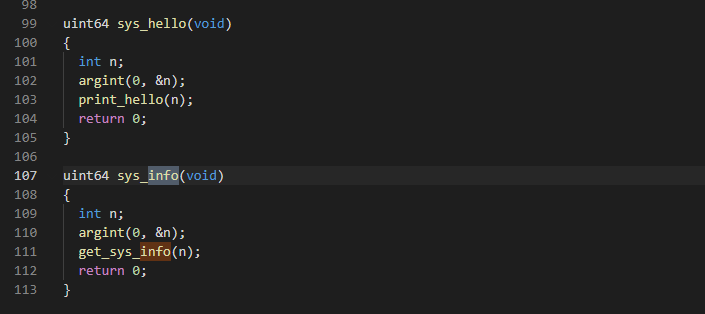


Modified syscall function to maintain count of total system calls made

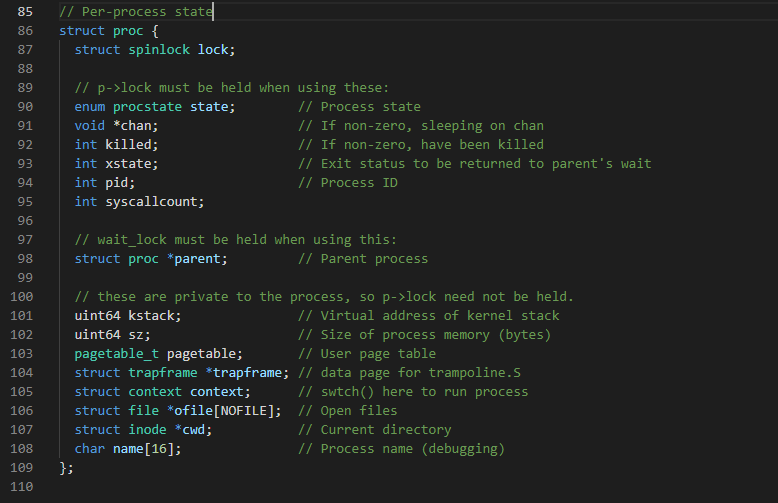


## Sysproc.c

Defined sys\_info syscall function and passed the argument to get\_sys\_info function for further processing



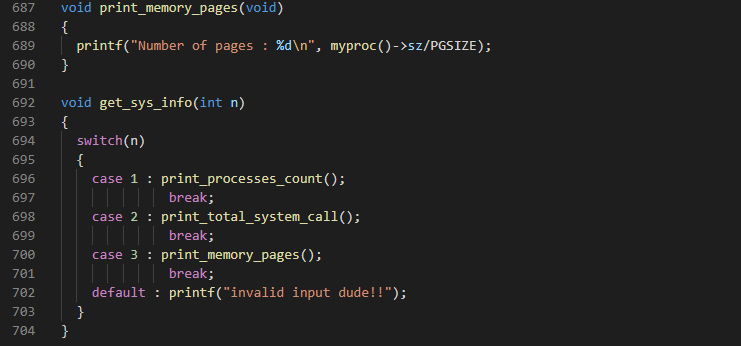
## Proc.h

Added a new structure variable syscallcount to keep count of system calls

## Proc.c

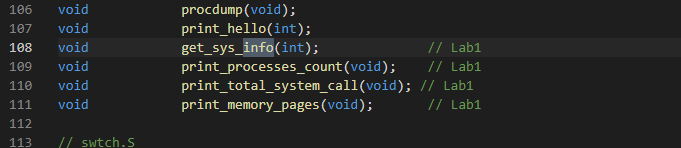
Defined new kernel functions





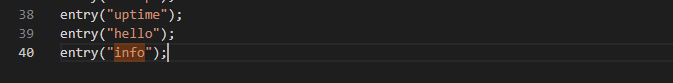
## Defs.h

Added entries for kernel functions



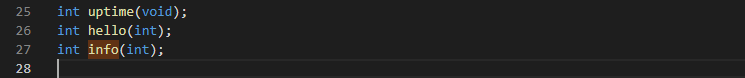
## Usys.pl

Added entry to update user syscall interface



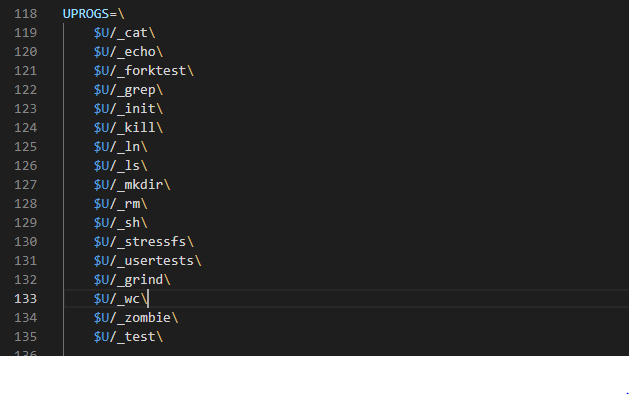
## User.h

Added entry to update user syscall interface. This would be the function our user program can call.



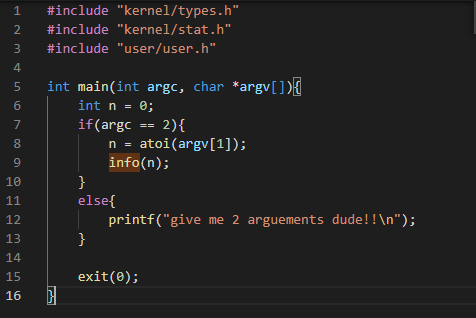
## Makefile

We add our user program to this file to make it available for xv6 source code for compilation.

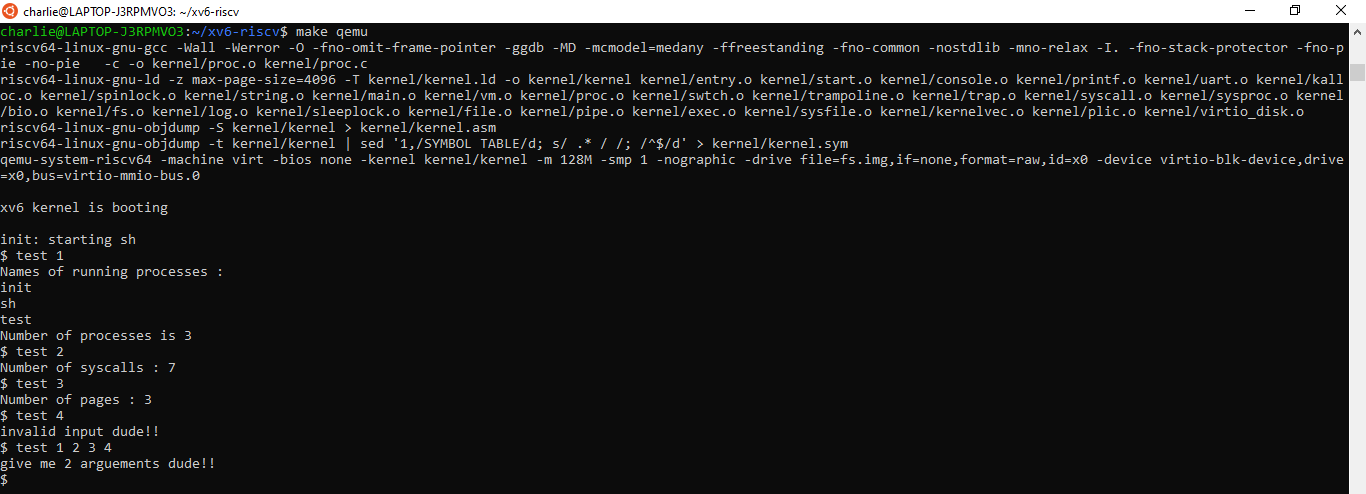


## Test.c

Created a user program test.c and added the main function. The user is expected to type in “test <case choice>”, so we check for the number of arguments to be 2. If not, we print an error message.



## Output



**Explanation of files**

First, the user program test.c is called. The code checks that 2 arguments are received (since expected user input is of the form “test <user\_choice>”) and passes it on to the function info.

In the kernel directory, we first define a new system call number for our new system call in the syscall.h file. Next, in the syscall.c file we add the pointer [SYS\_info] to the function pointer array. This uses the newly added system call number to point to system call. The corresponding function is declared in this file as well. We have incremented the variable syscallcount in the syscall function that will be called each time a new system call occurs, to keep track of this count.

In the Sysproc.c file, we defined our system call functionality. A new function get\_sys\_info is invoked from here which handles all the cases.

The proc.h file has a structure that contains variables that store data related to each process. We have declared the new variable syscallcount here to bind it with processes.

In proc.c, get\_sys\_info and all other required functions are created and implemented. We created individual functions for each implementation.

1. Case 1 - Count number of running processes ( print\_processes\_count(void))

xv6 initializes an array (with length 64, defined as NPROC) of processes with state as UNUSED. All valid states are defined in procstate enum.

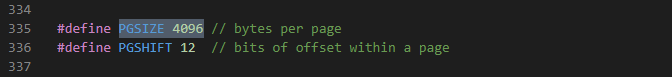
Whenever a new process is created, the next available index in the array is updated with the process structure. Before reading the process array, we acquire a lock on it, to make sure other CPU’s don't grab the same data. Xv6 provides functions to acquire and release a lock-on process in the spinlock.c file. We then loop through this array, count the number of processes which don't have state as UNUSED, and print them. Along with printing the count, we are also printing the process name. The lock is then released.

1. Case 2 - Count system calls made( print\_total\_system\_call(void))

All system calls made in xv6 go through the function syscall(void) defined in syscall.c. This function handles all the system calls from user code. We simply create a new integer field in our process structure called syscallcount (in file proc.h). Then, we increment this field in the function syscall. To print this to the user, we access our running process using the myproc function provided by xv6 in proc.c and print the value of syscallcount.

1. Case 3 - Count number of pages used by our process( print\_memory\_pages(void))

We access the sz variable in the process structure which gives us the entire size of the process and divide it by the page size PGSIZE of the system to get the number of pages. This PGSIZE is defined as 4096 bytes in the riscv.h file as shown below:



In defs.h we added a declaration for all of our function definitions to be able to use it moving ahead. Now we edit the usys.pl and user.h file to make the system call available for the user to use. In usys.pl we simply add an entry “info”. Similarly, we add an entry for our system call in the user.h file as well. On adding this, the system will map any system call to this function to the system call number defined earlier.

User and Kernel mode changes :

When our test.c user program calls the info function, a system call with number 23 is called. This is the point where the mode changes from user mode to kernel mode. Next, the function pointer sys\_info (in syscall.c) which has the index SYS\_info or 23 will call the system call function. The function, defined in sysproc.c is executed and invokes our defined kernel functions in proc.c. These functions are executed and the required result is printed by these functions. The mode is now changed from kernel mode to user mode and the program is then completed with an exit statement.