**CS202 - Advanced Operating System**

**LAB 1**

* Tejas Milind Deshpande , sid- 862393211
* Satya sri Nandan Paritala , sid - 862395423
* Nunna Lakshmi Saranya , sid- 862394536

**A.Demo Video**

[**https://drive.google.com/file/d/1mFmCdxONukmK4wqBIbP9nu593\_mp86YM/view?usp=sharing**](https://drive.google.com/file/d/1mFmCdxONukmK4wqBIbP9nu593_mp86YM/view?usp=sharing)

**B.List of all modified files-**

* MAKEFILE
* kernel/defs.h
* kernel/proc.c
* kernel/proc.h
* kernel/syscall.c
* kernel/syscall.h
* kernel/sysproc.c
* kernel/kalloc.c
* user/user.h
* user/usys.pl
* New file for testing - user/test.c

**C. Changes and screenshots-**

As per the requirement of the assignment, changes were made to the files.

**Part 1: sysinfo**

(We used printinfo for the name of the function)

The program takes 1 integer that can be 0,1 or 2 and prints according to the user’s information requirement.

If it is 0 - The system call returns , the number of system’s active process ( it can be running,

ready , waiting or zombie )

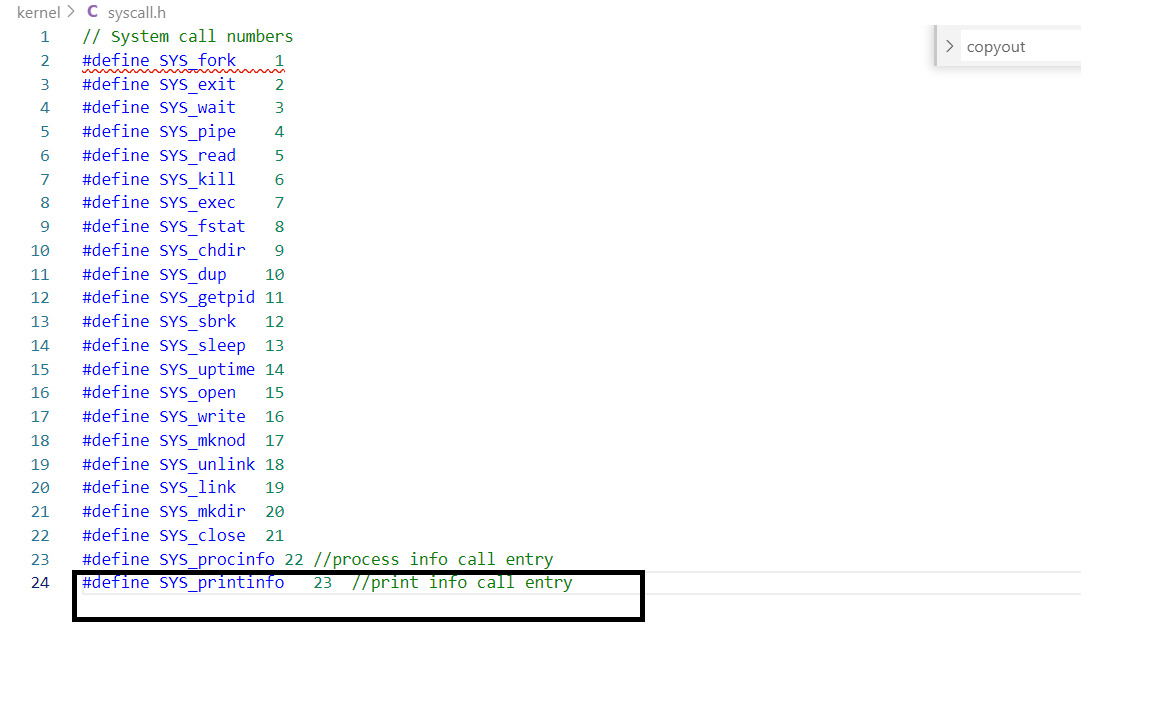
If it is 1 - The system call returns, the number of system calls made since boot up.

If it is 2 - The system call returns, The number of free memory pages in the system.

Otherwise returns -1

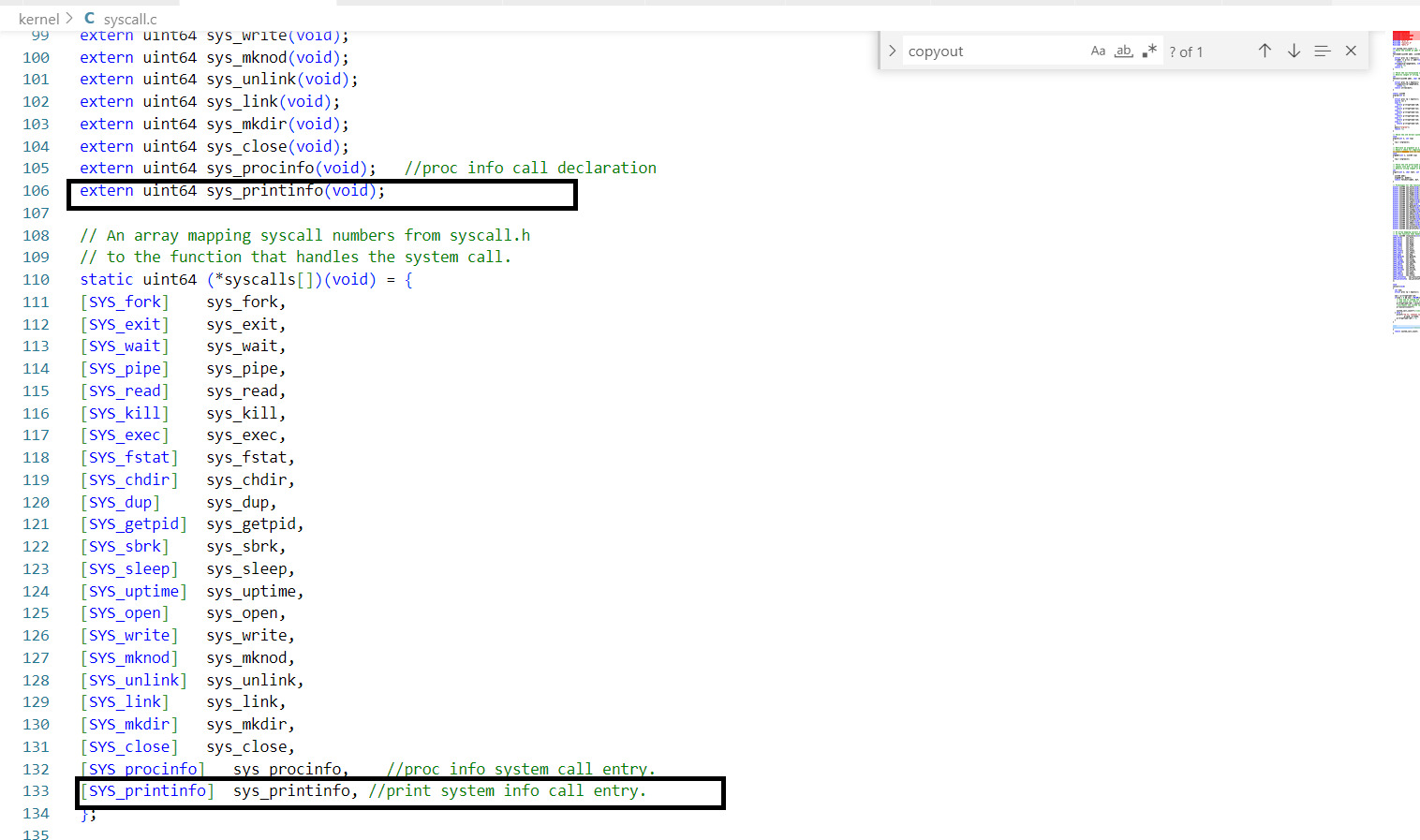
**syscall.h**

A new SYS\_printinfo variable is created and assigned to number 24

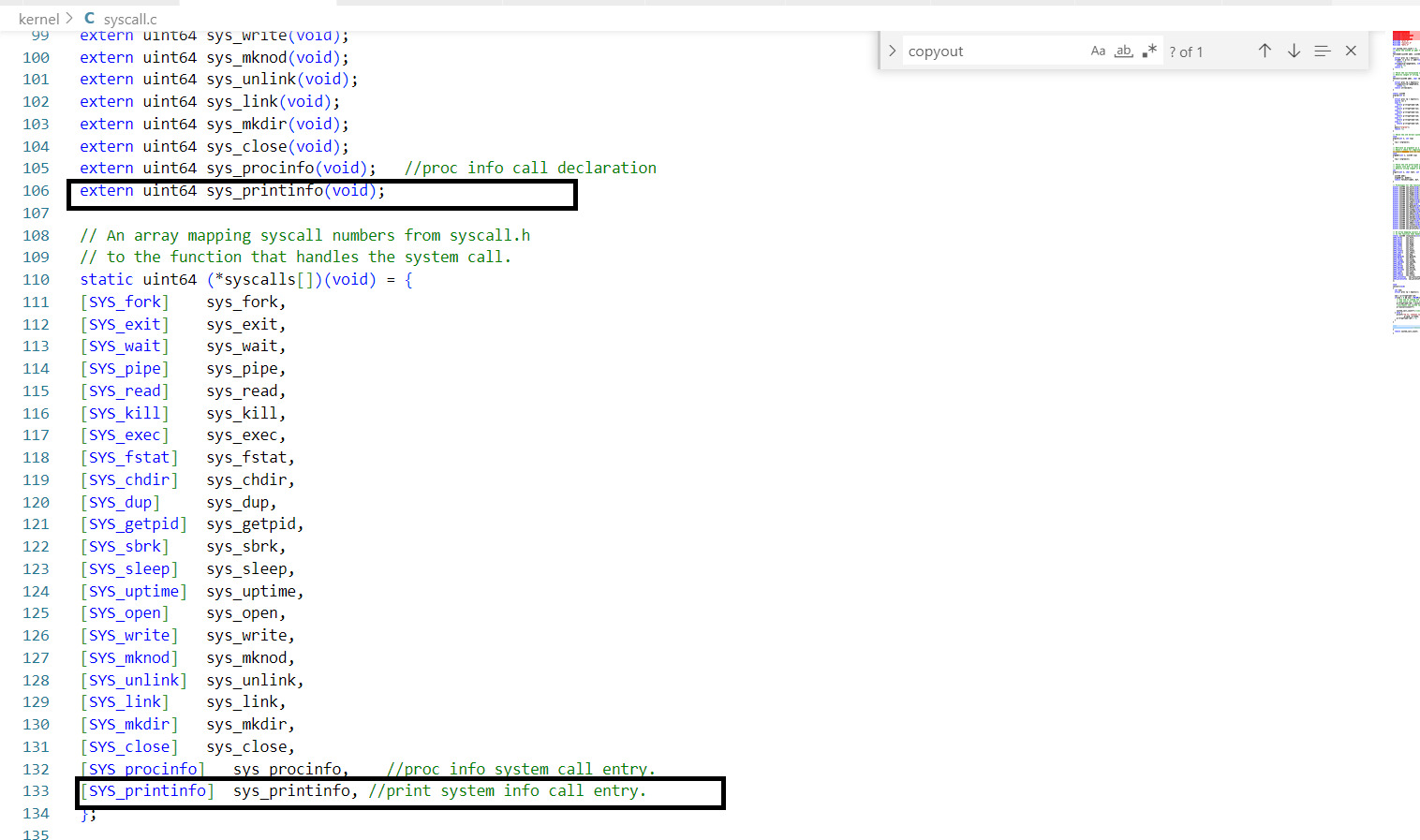


**syscall.c**

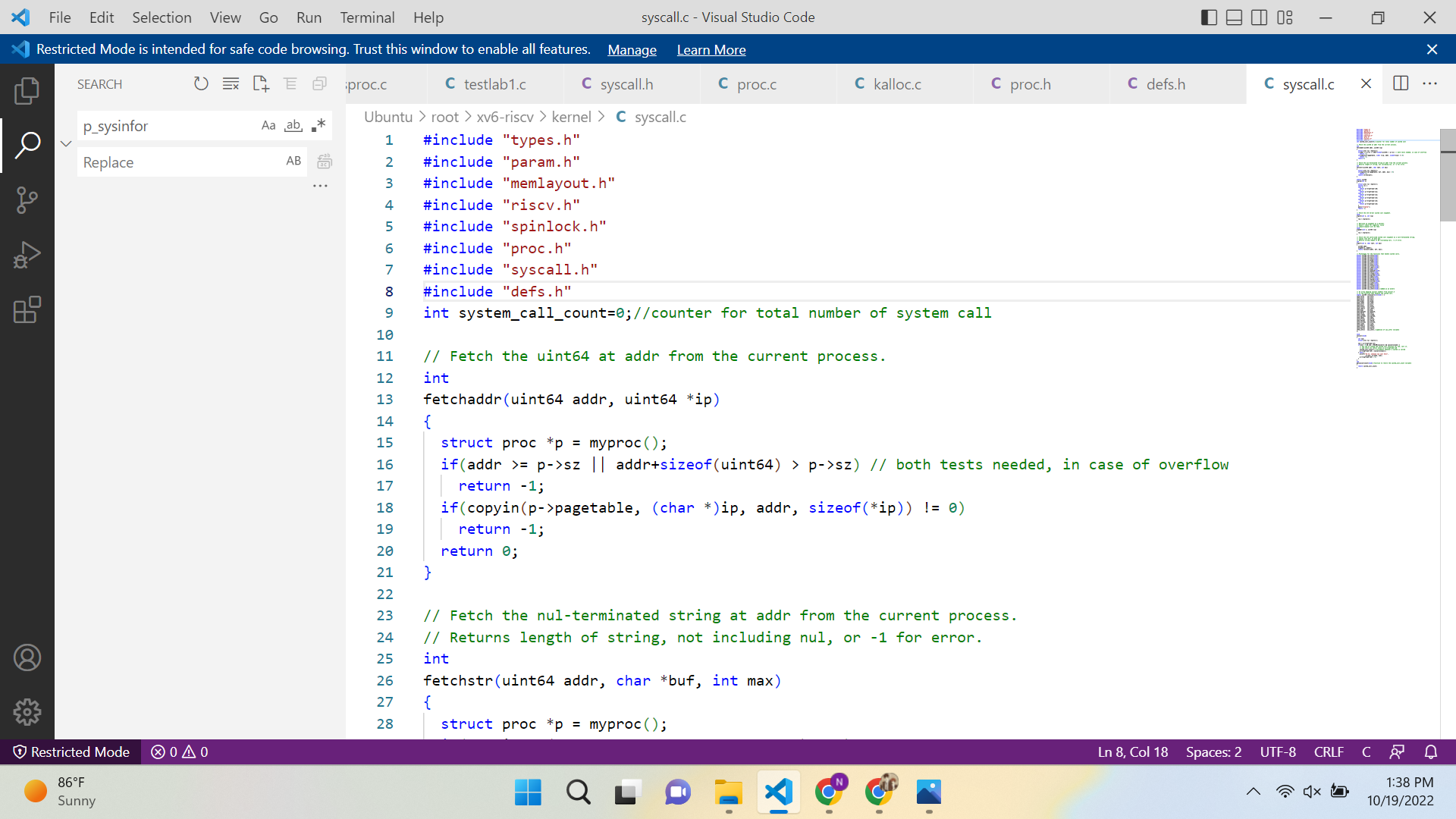
In order to initiate the system call from outside the kernel , we make the sys\_printinfo call the extern.



Updated this file with the pointer.



The existing syscall function is modified such that it increments the system\_call\_count variable whenever a system call is initiated.



:

:



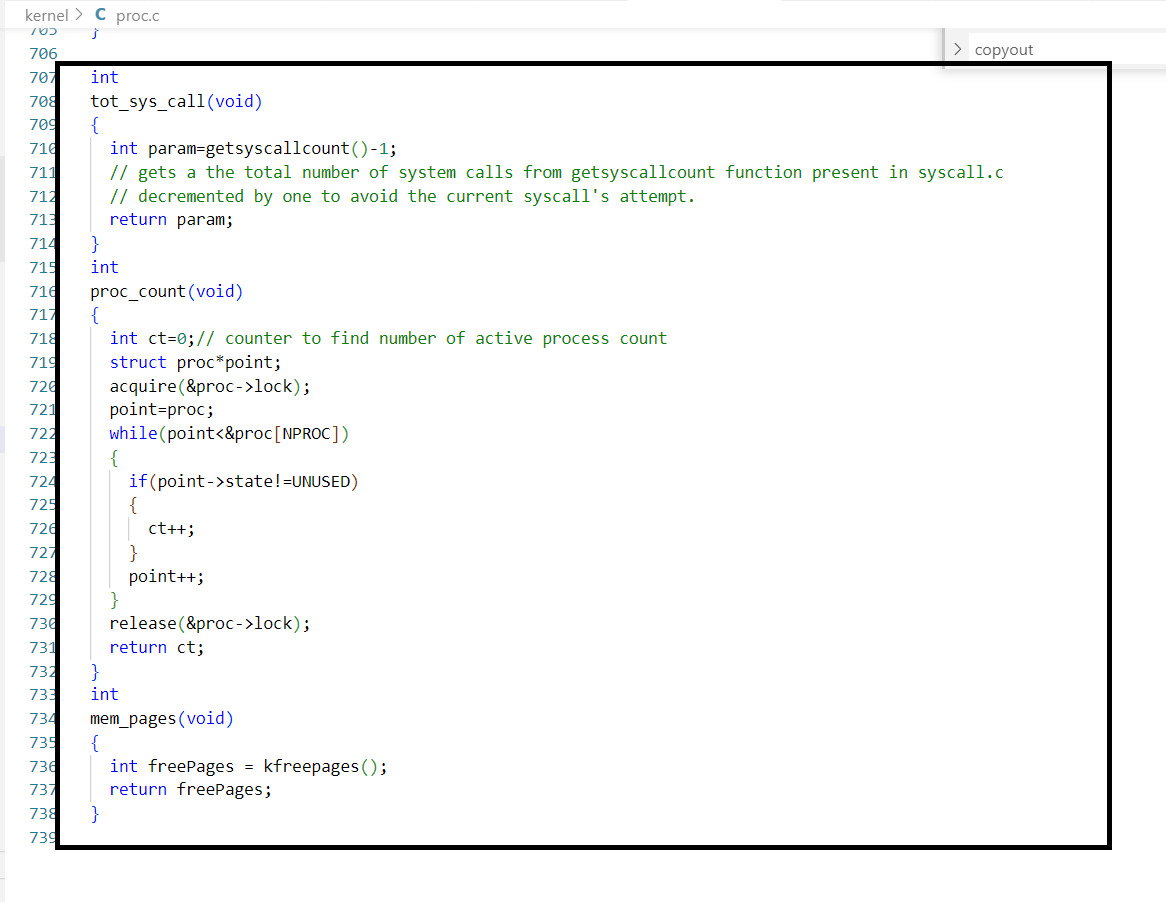
A new function is introduced to get the system call count



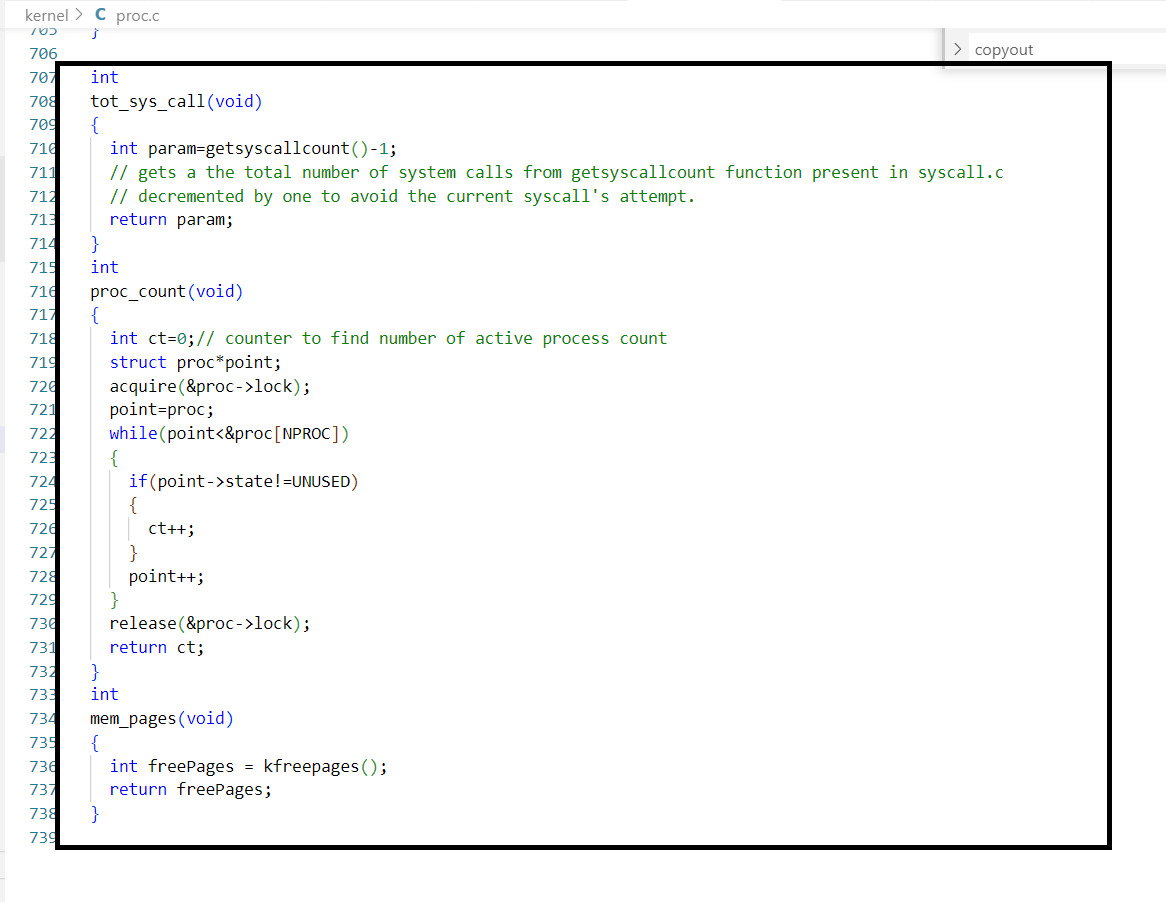
**proc.c**

User can request the total number of processes or or total number of system calls or total number of free pages. Each has a new kernel function.

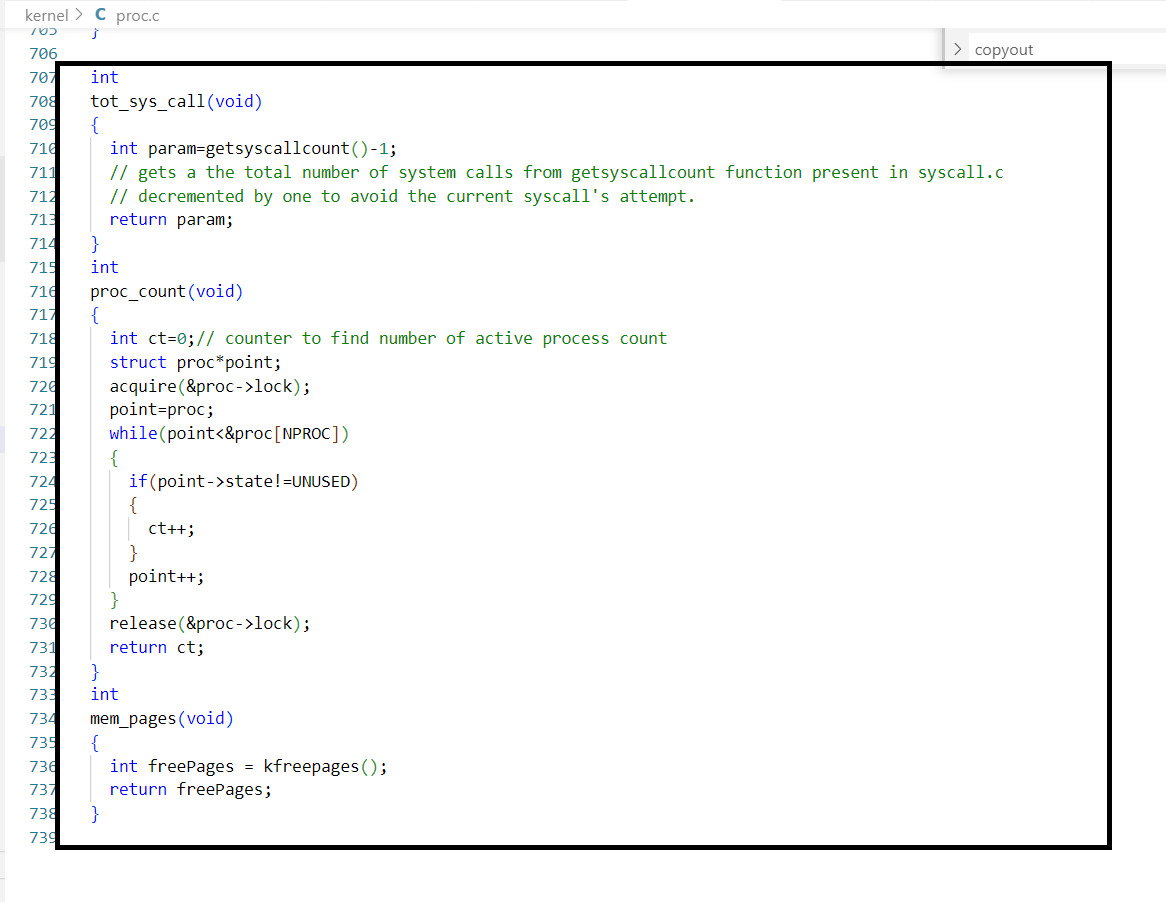
* Function tot\_sys\_call() - to return the total number of system calls since boot up



* Function mem\_pgs() - to return the number of free pages present.



* Function process\_count()- to return the number of active processes in the system.



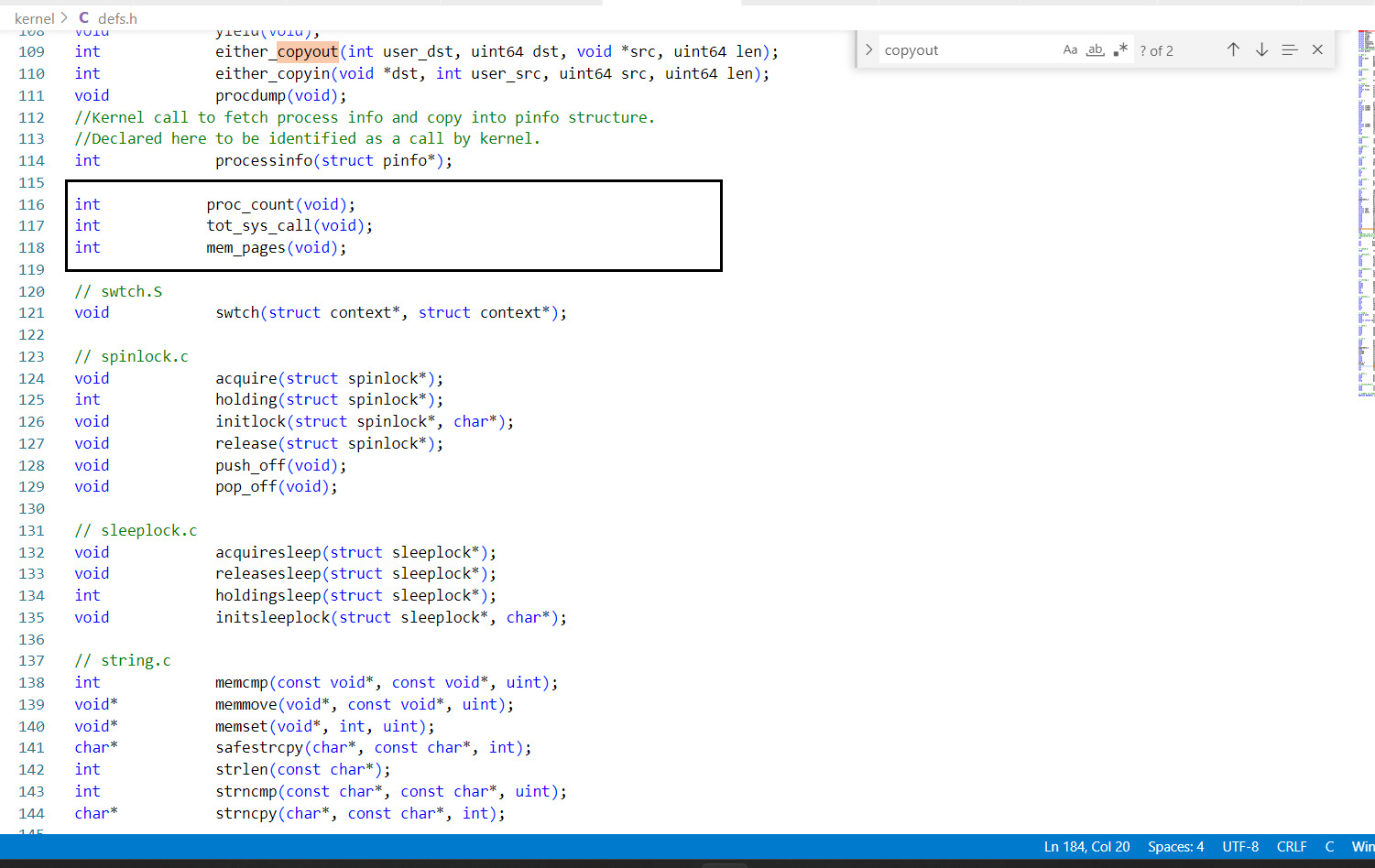
**kalloc.c**

New function to get the count of free memory pages.

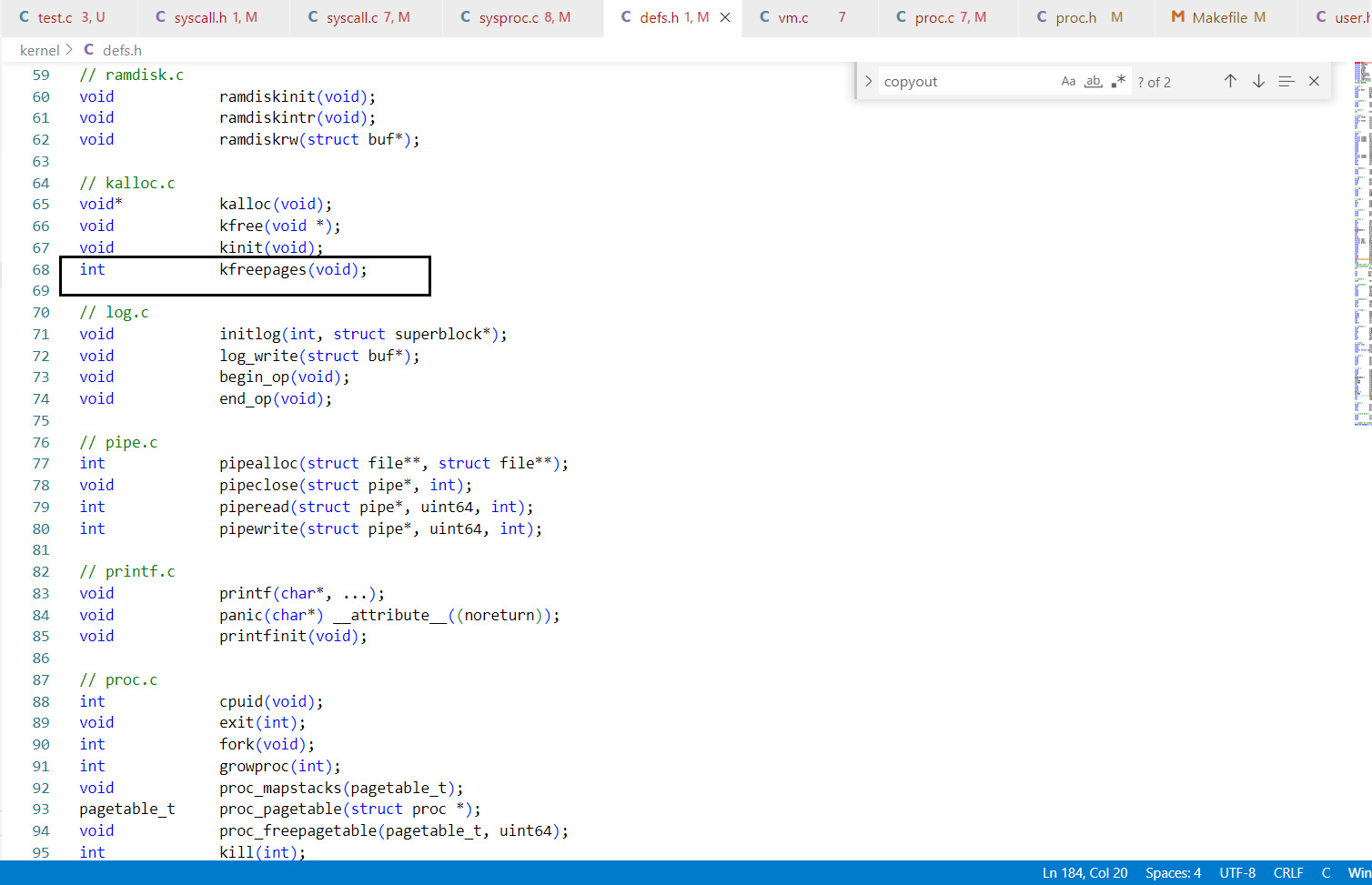


**defs.h**

All the new functions were added.

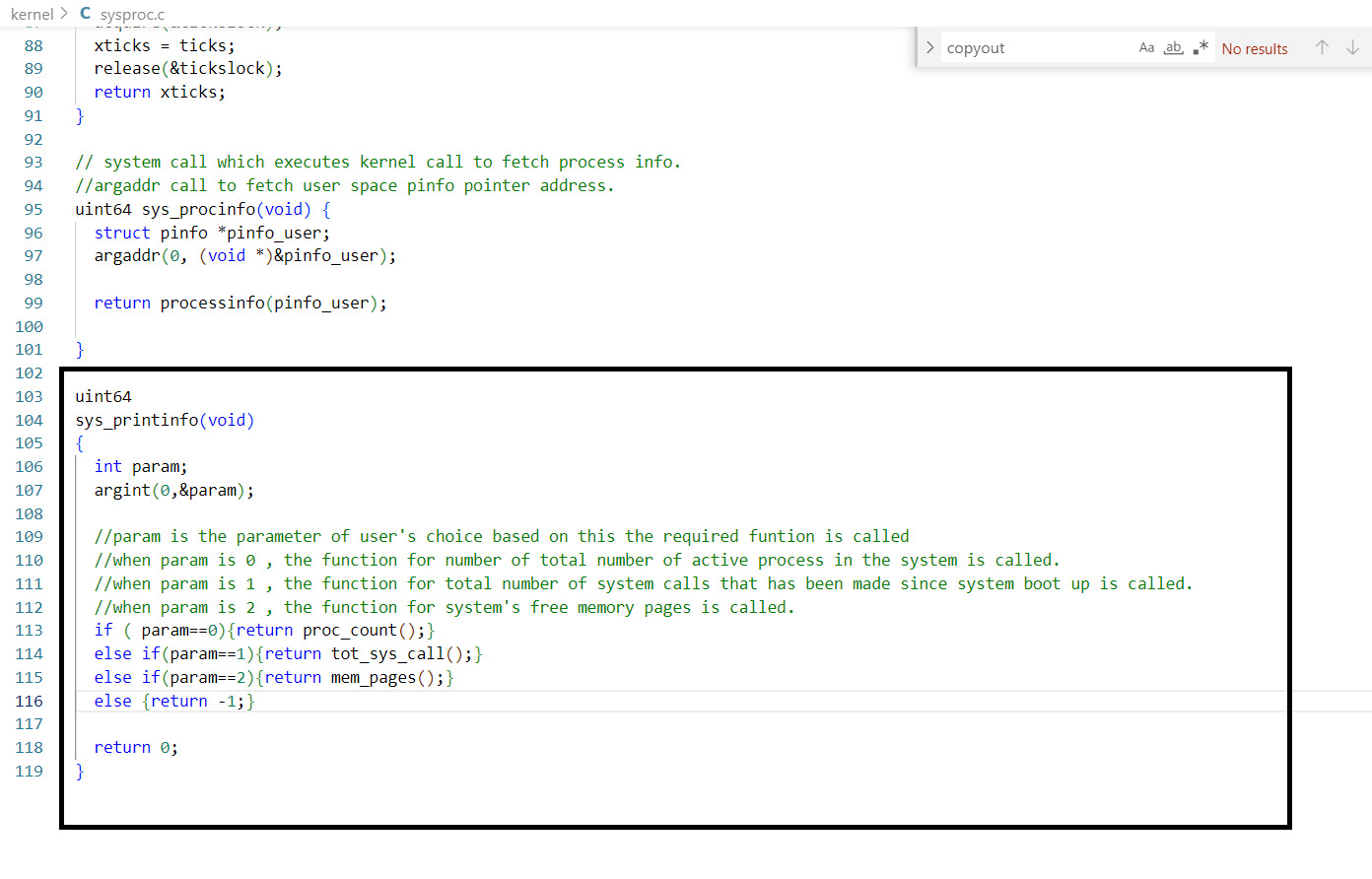






**sysproc.c**

A new sys\_printinfo() function is created which takes an argument and based on it calls respective functions in proc.c



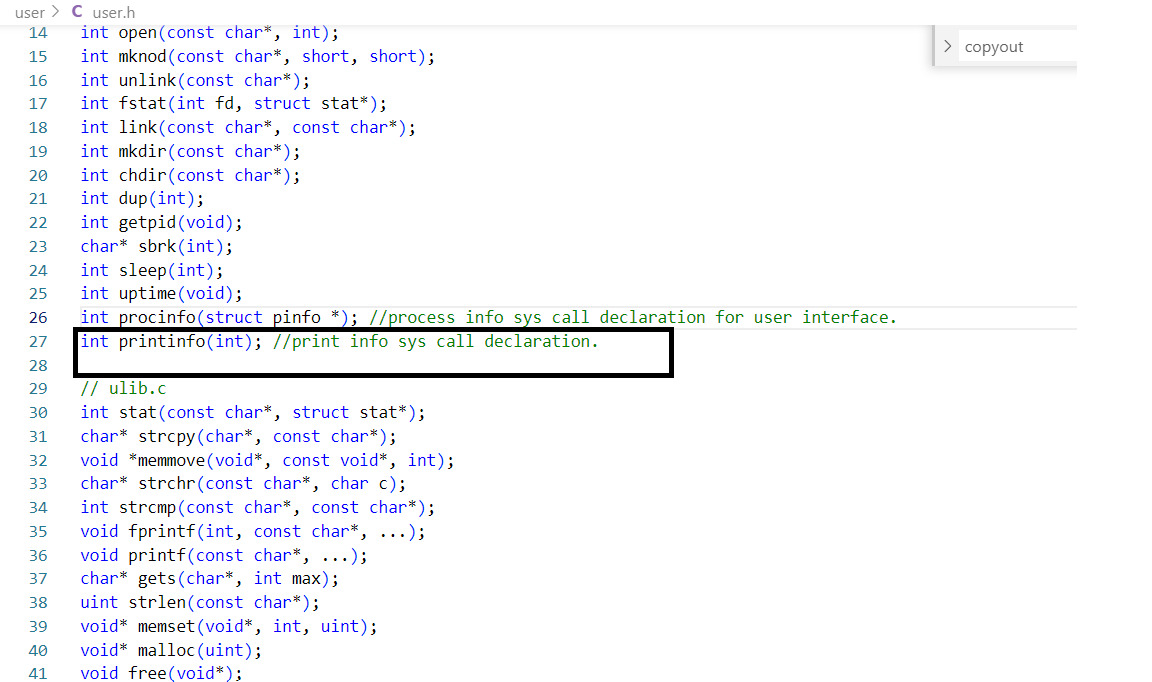
**usys.pl**

Created system call interface, was added as an entry to this file.



**user.h**

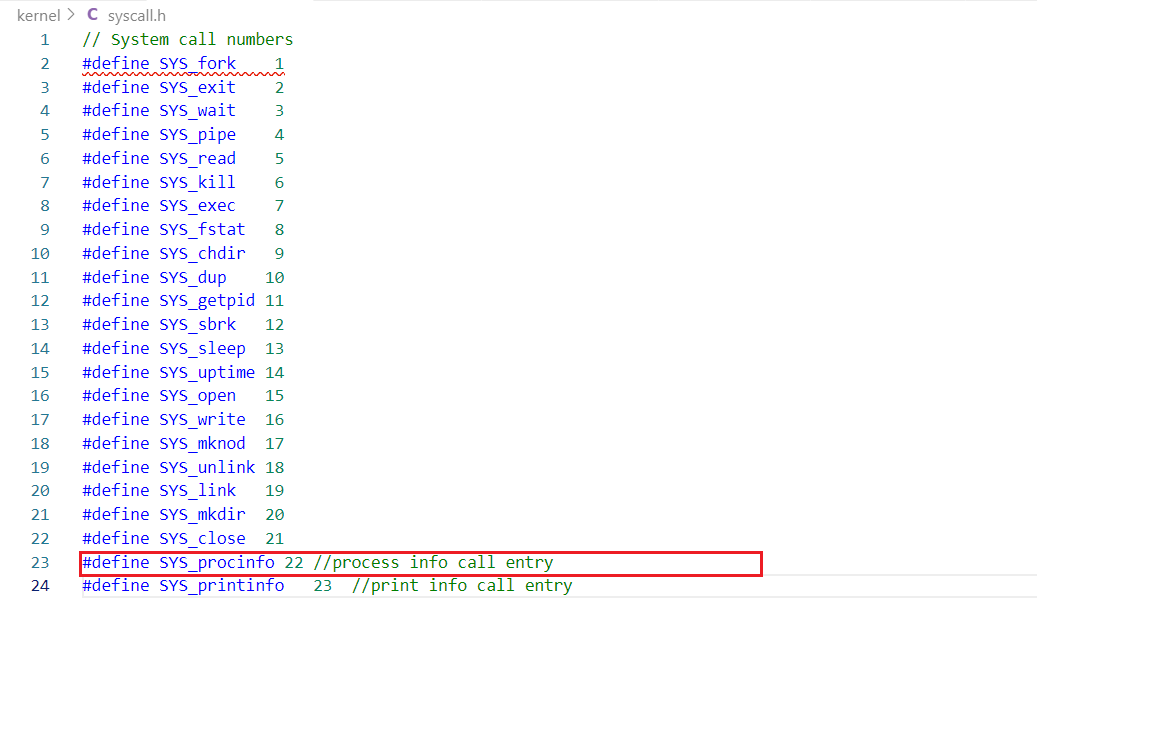
The created interface was updated in user.h file.



**Part 2: Proc Info**

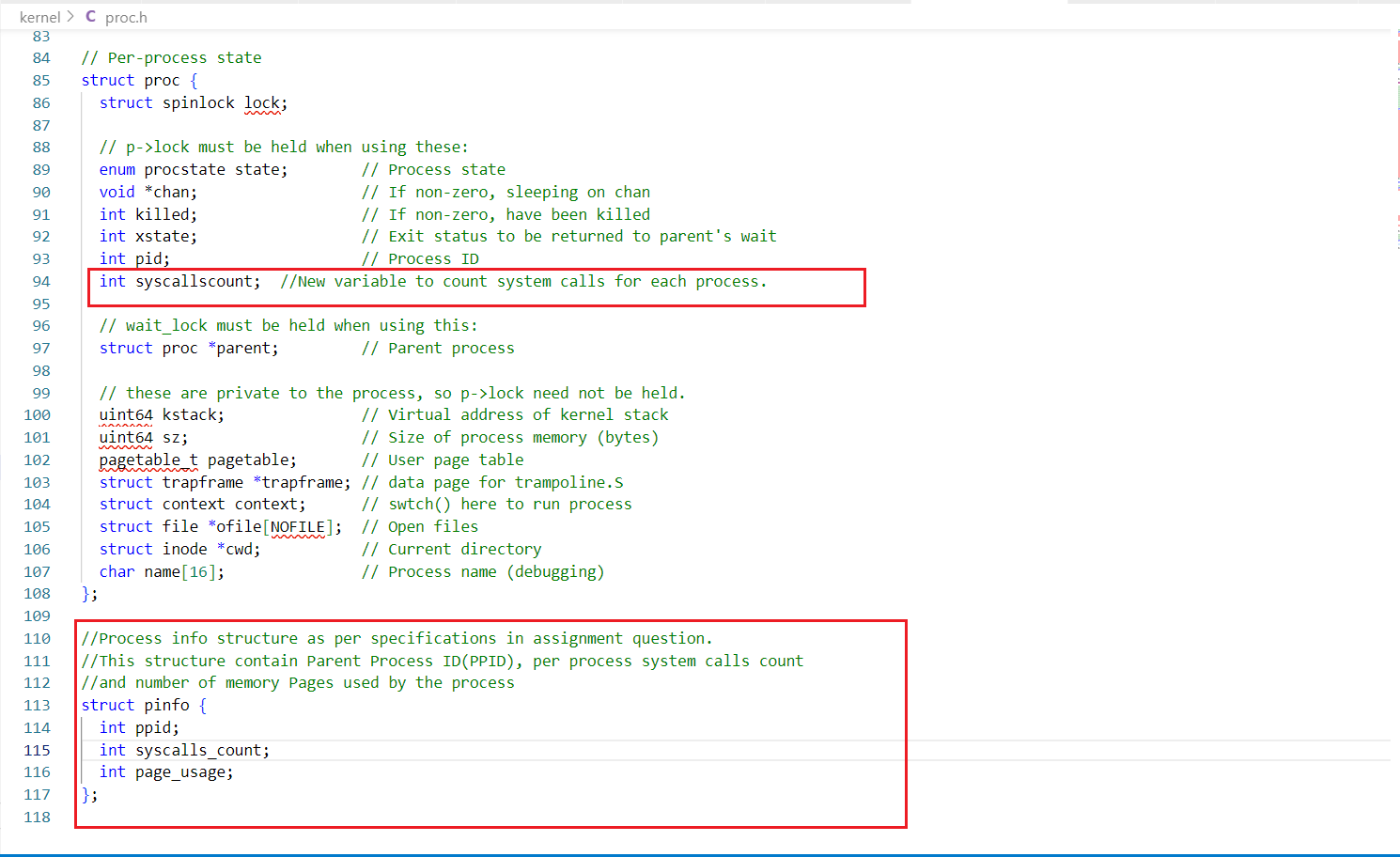
1. “**syscall.h”**

Defining new system call number for procinfo:

****

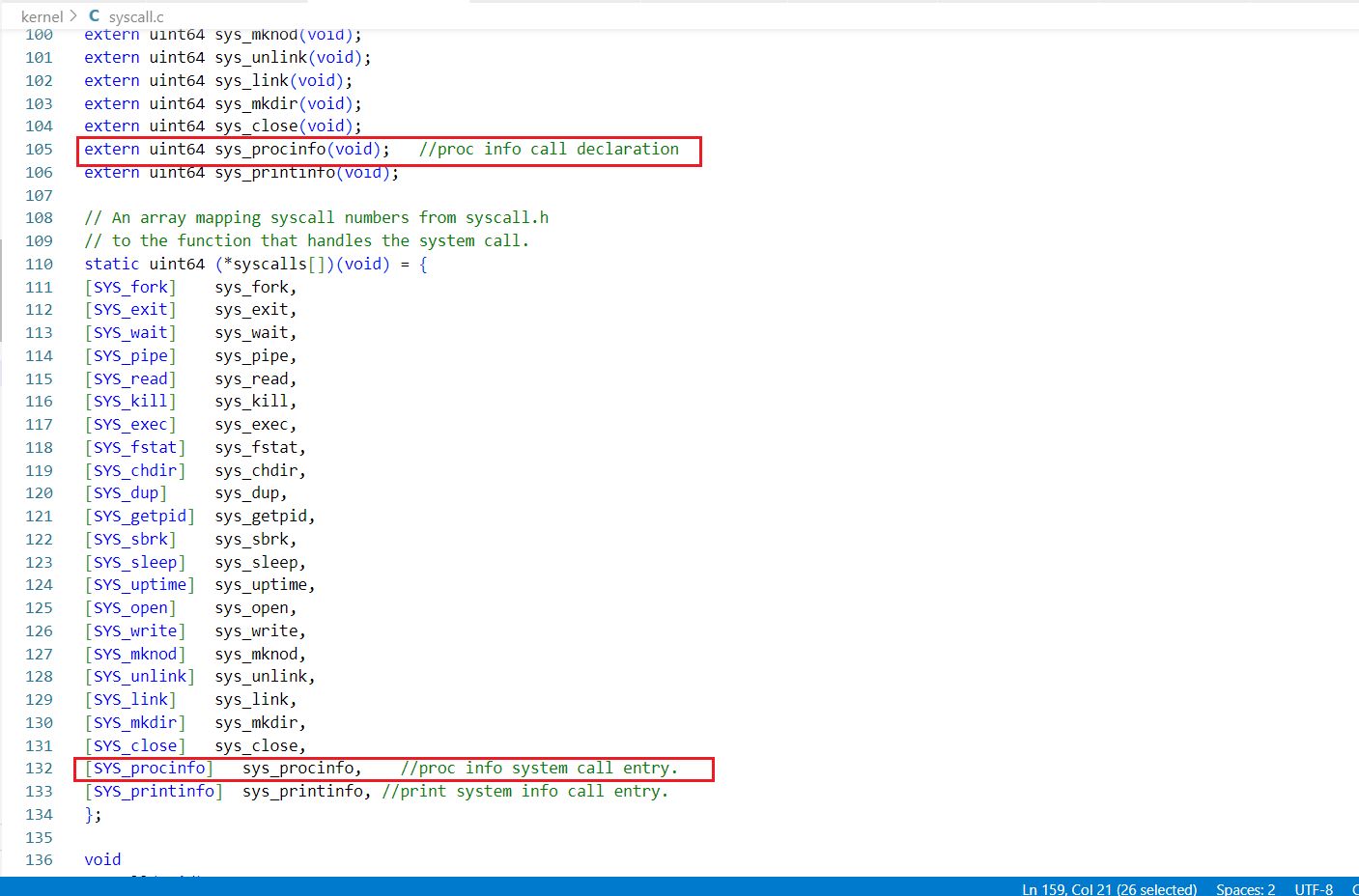
1. **Kernel/proc.h**

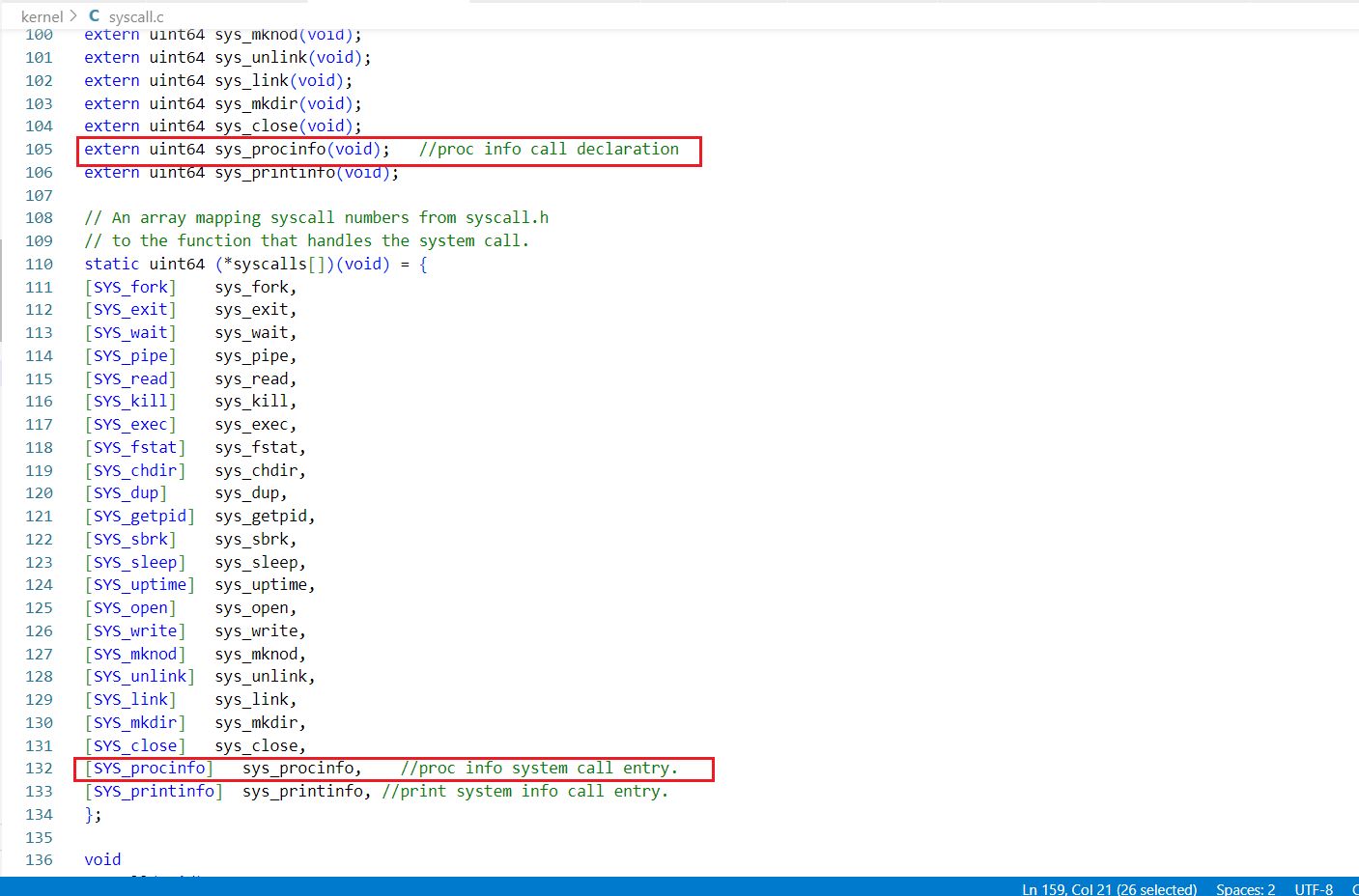
Declaring new **system call count variable** in the PCB structure for each process and defining the **pinfo** structure in the kernel space for kernel reference:

****

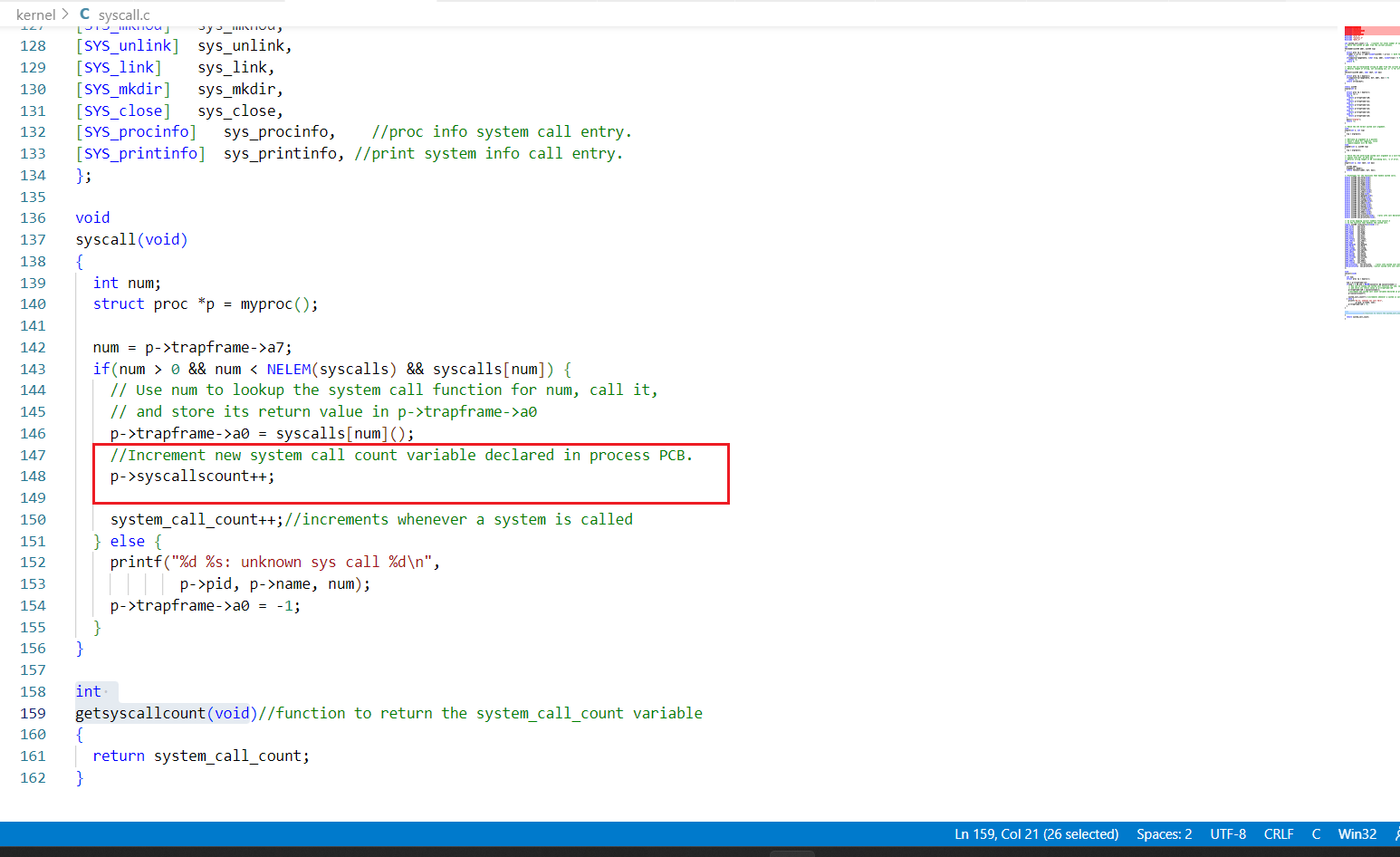
1. **syscall.c**

Updating system call table by adding entries for procinfo call:

****

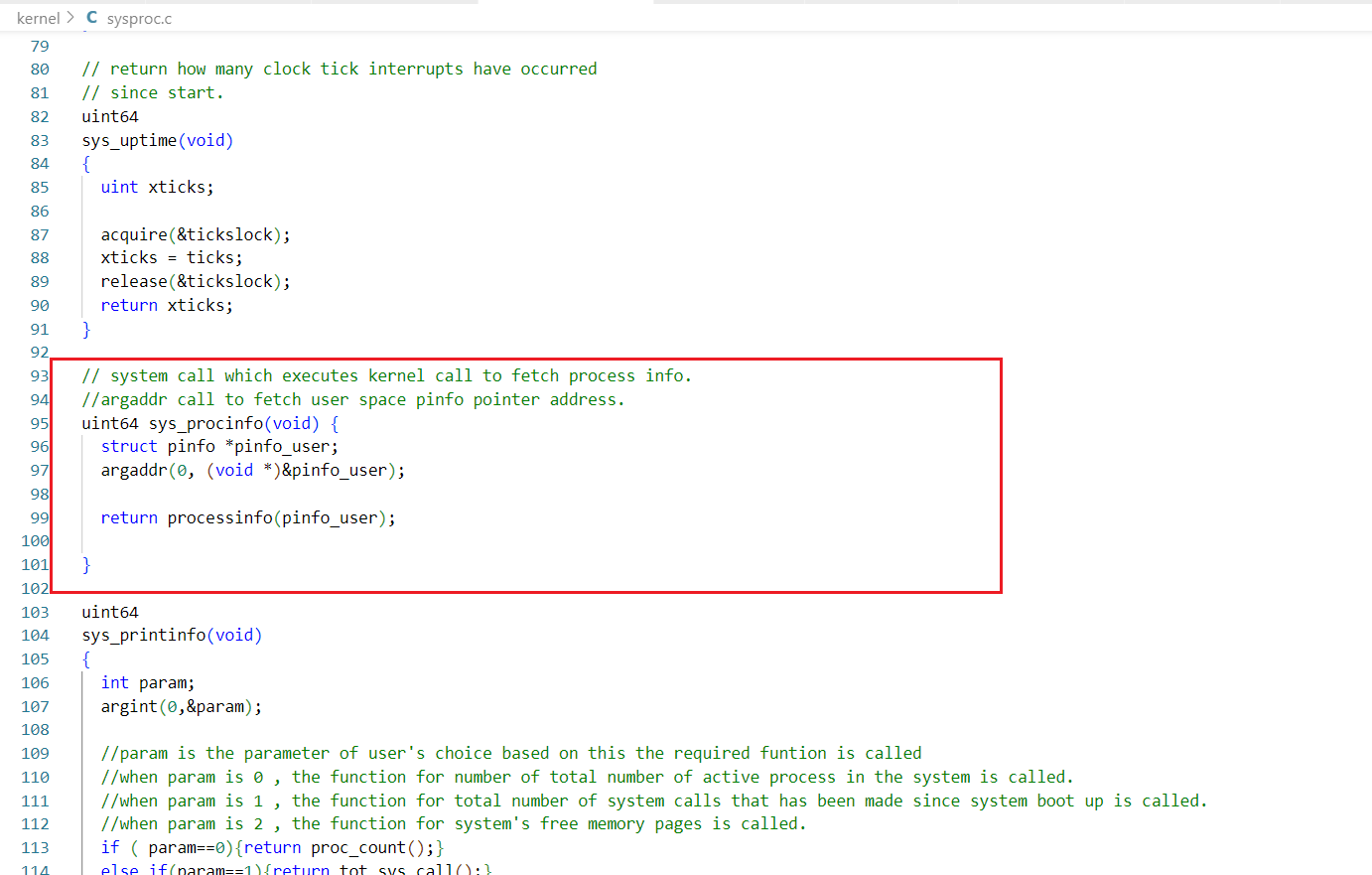
****

Incrementing the new system call count variable defined in the current process PCB:

****

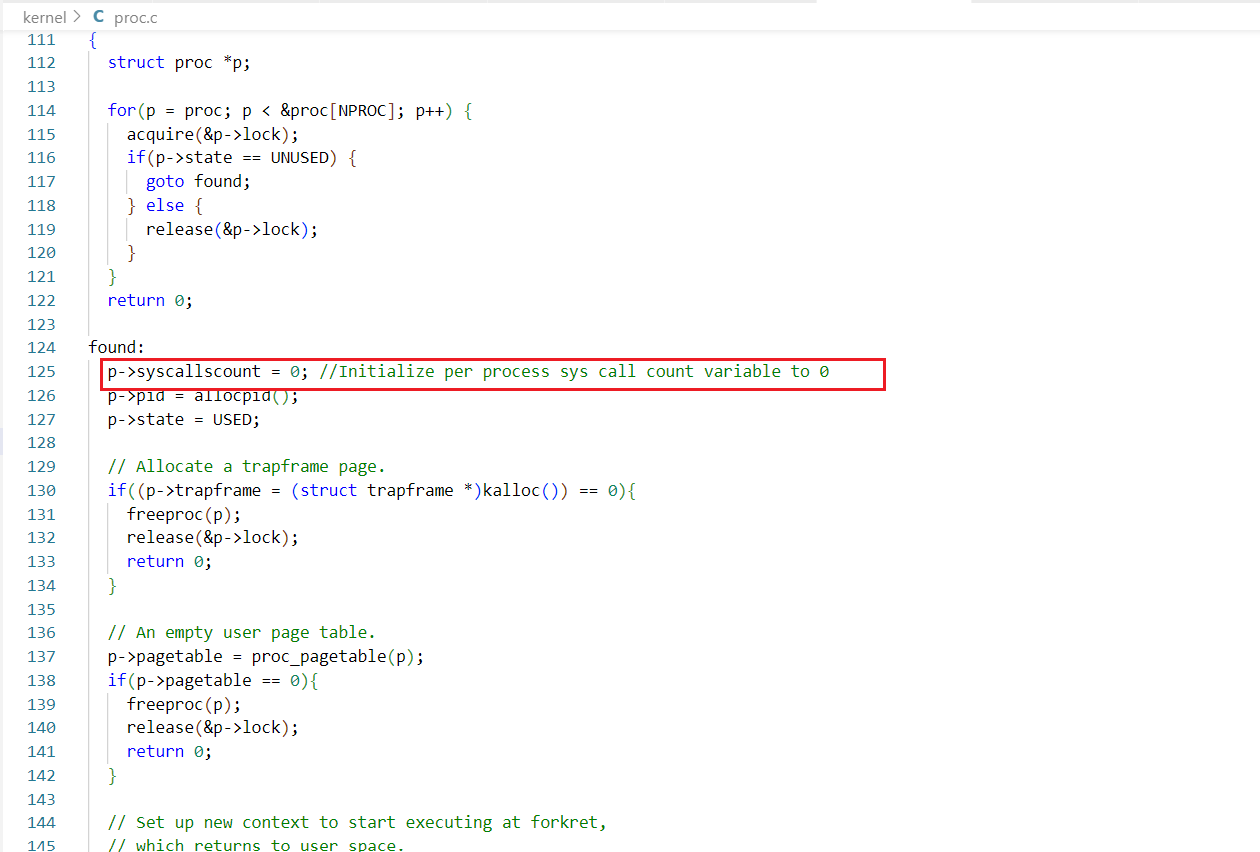
1. **“sysproc.c”**

**Creating a new procinfo system call :** This procinfo call will be executed from user space. We are retrieving a pointer to the user space pinfo structure using the **argaddr**() function and we are forwarding this pointer to the **processinfo** kernel call.



1. **“proc.c”** :

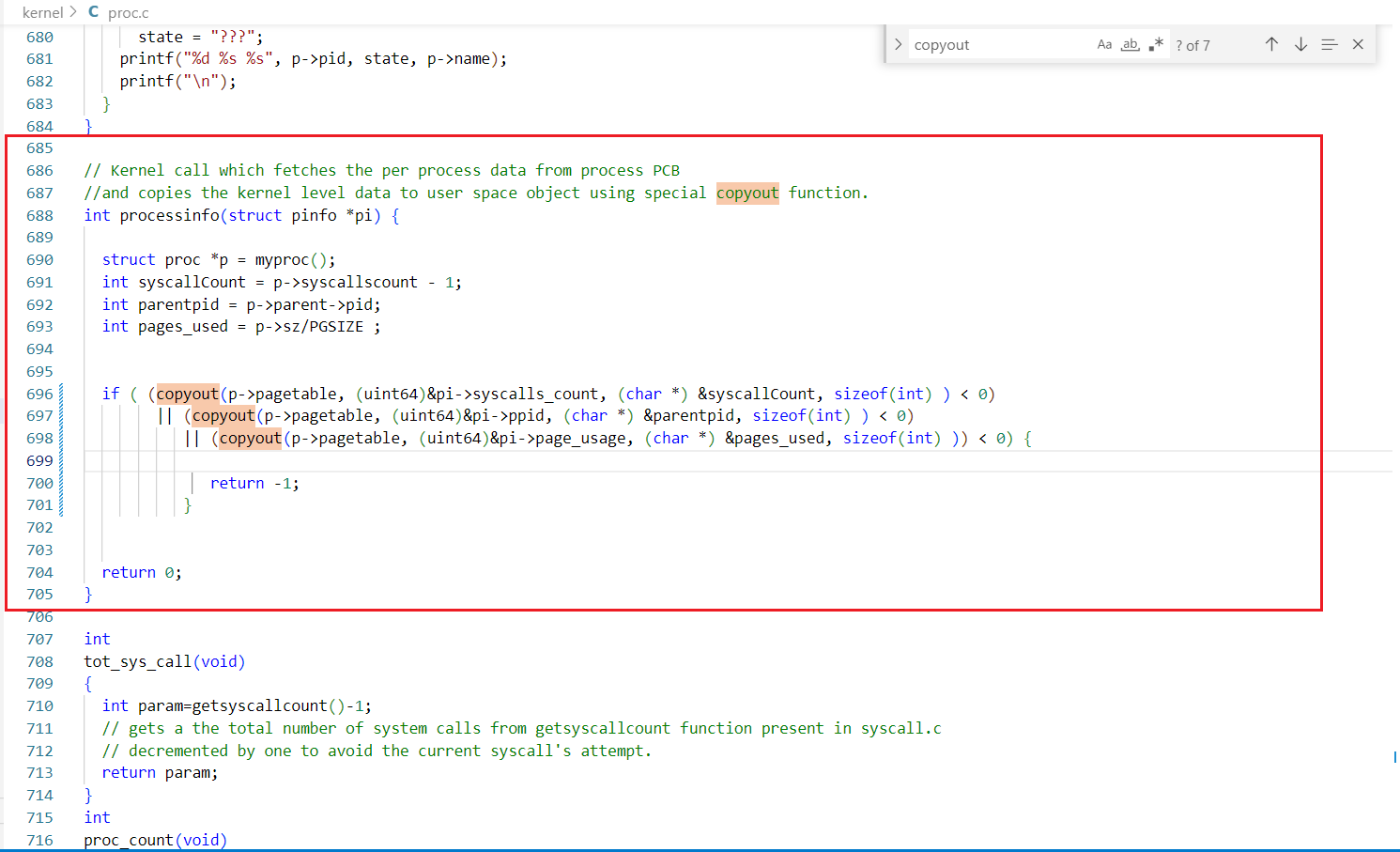
Initialize the **new per process system call count** to 0 in the Process Control Block:

****

**Creating a new printinfo kernel call:** In this call, we are fetching the Process Control Block(PCB) of the current process. From this PCB, we are fetching the number of system calls executed, the parent process ID(PPID) and the amount of memory usage in pages for the current process.

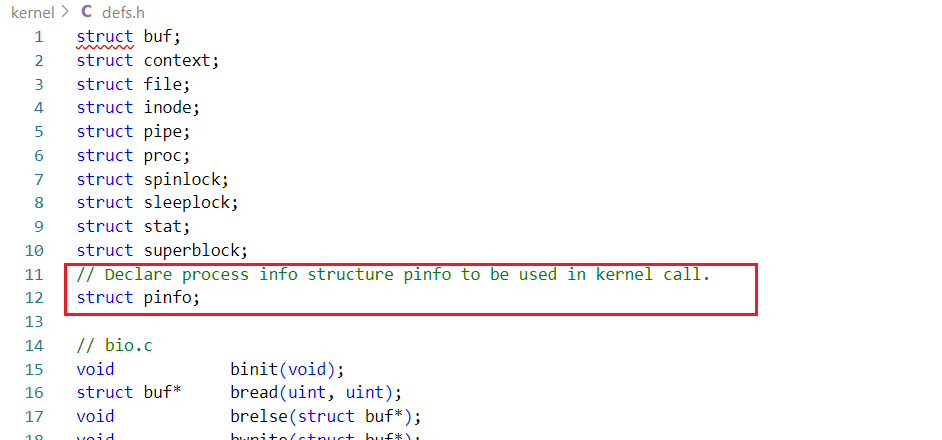
After fetching the above information, we are using a special kernel function called **copyout** for copying the data into the user-space pinfo structure. We are receiving a pointer to the user-space pinfo structure, which we pass as copy destination in the copyout call.

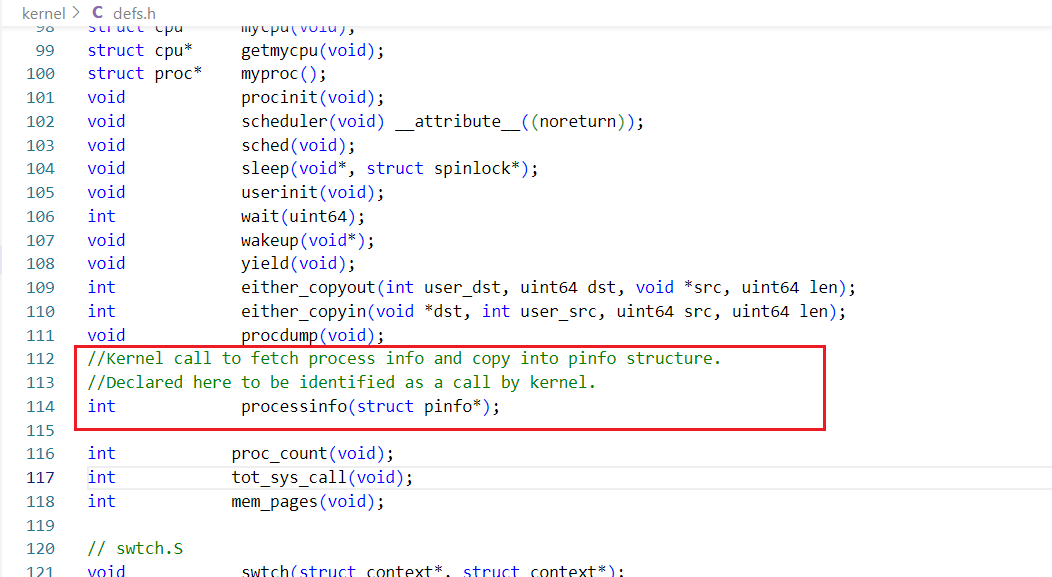
If all the copyout functions are executed successfully, then we return 0 as successful execution or we return -1 in case on failure.



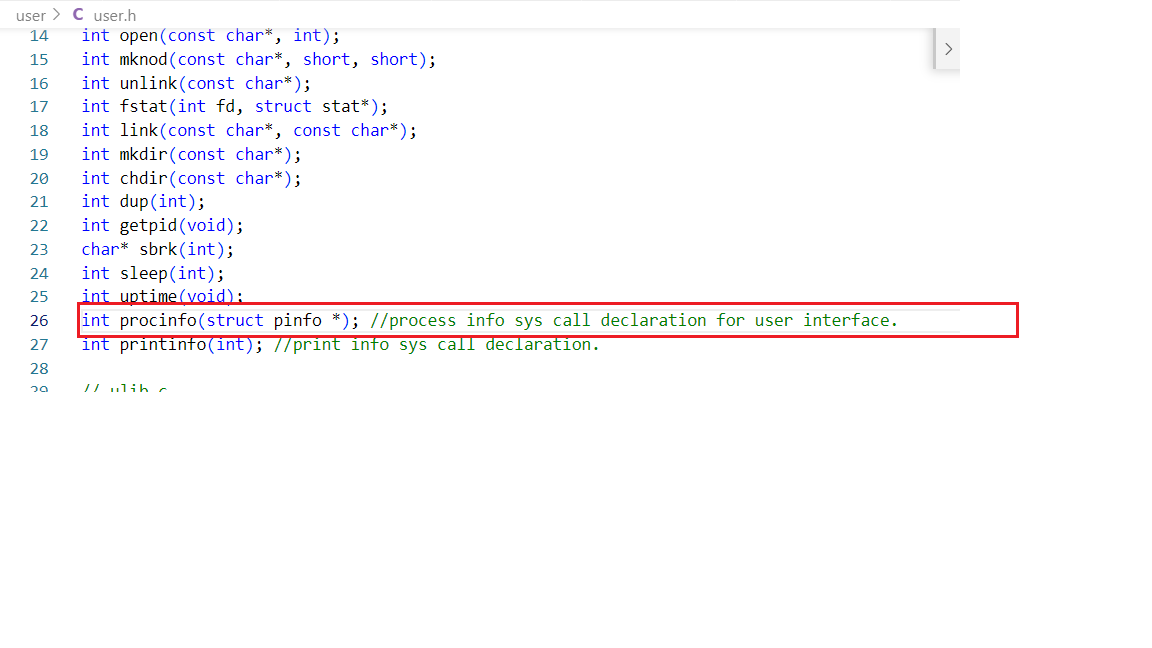
1. **Declare kernel function in “Kernel/Defs.h”:** Here, we declare the pinfo structure, since we are passing an argument of type pinfo in our declared method.

We declare the processinfo method here so that the system registers it as a kernel call.

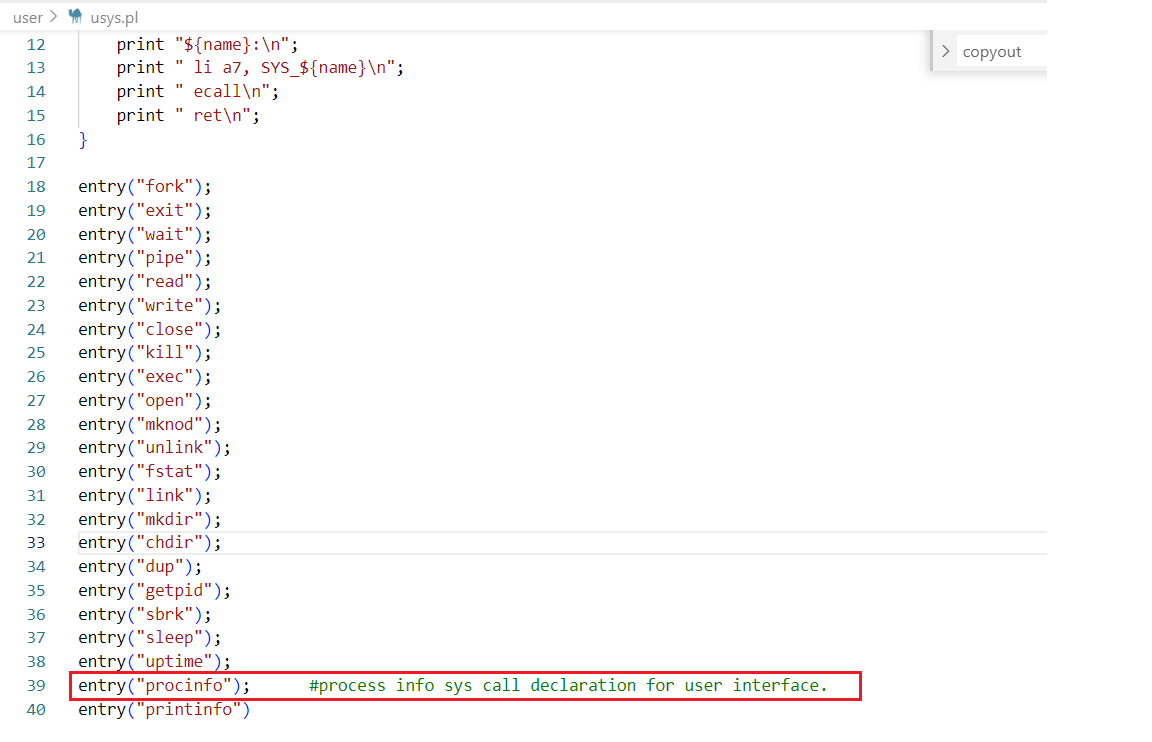




1. **Adding system call in user interface in “user.h”:** This helps the system map the entry into system call when we execute it from the user space.

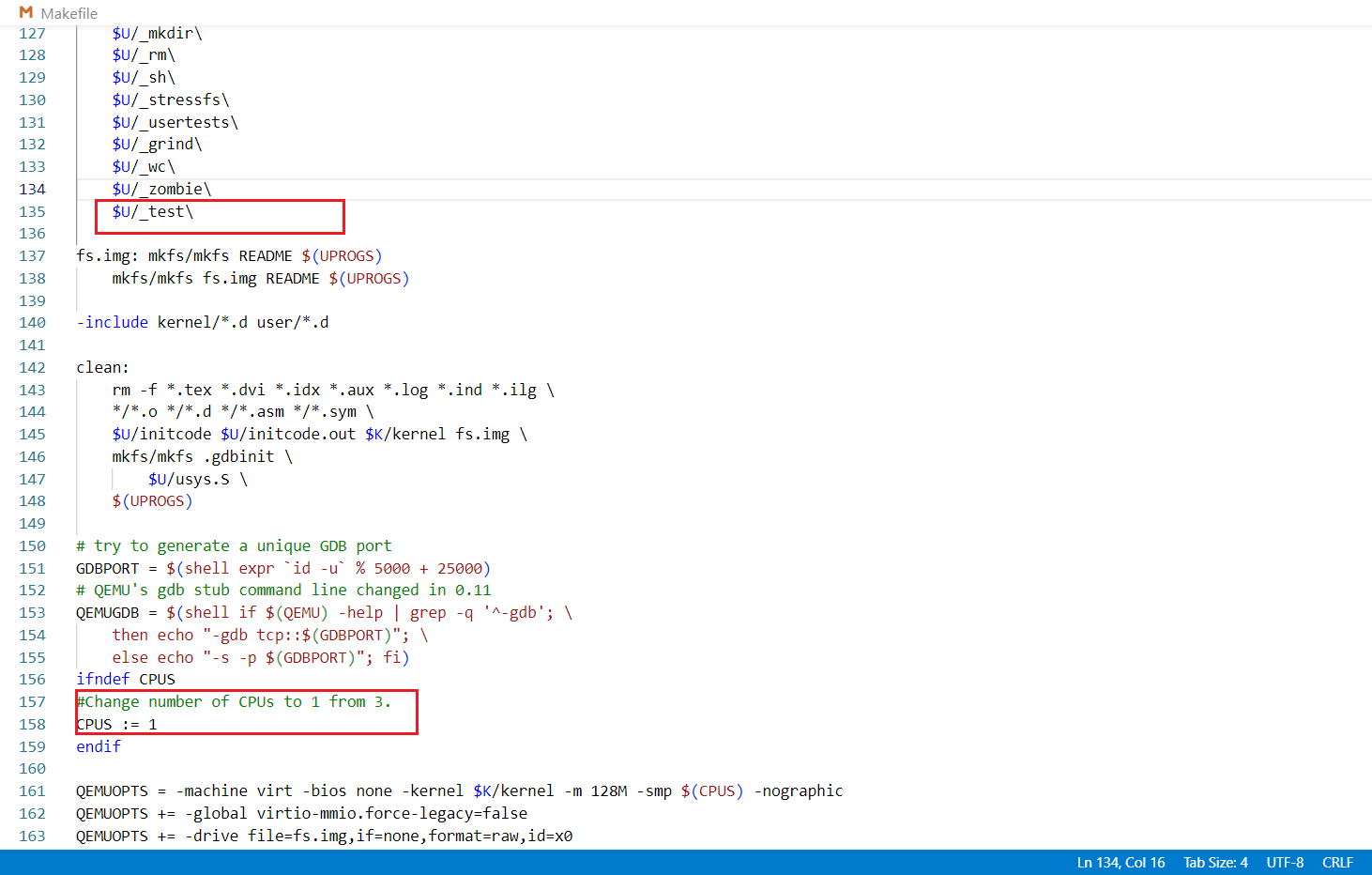
****

1. **Adding system call in user interface in “usys.pl”:** This helps the system map the entry into system call when we execute it from the user space.

****

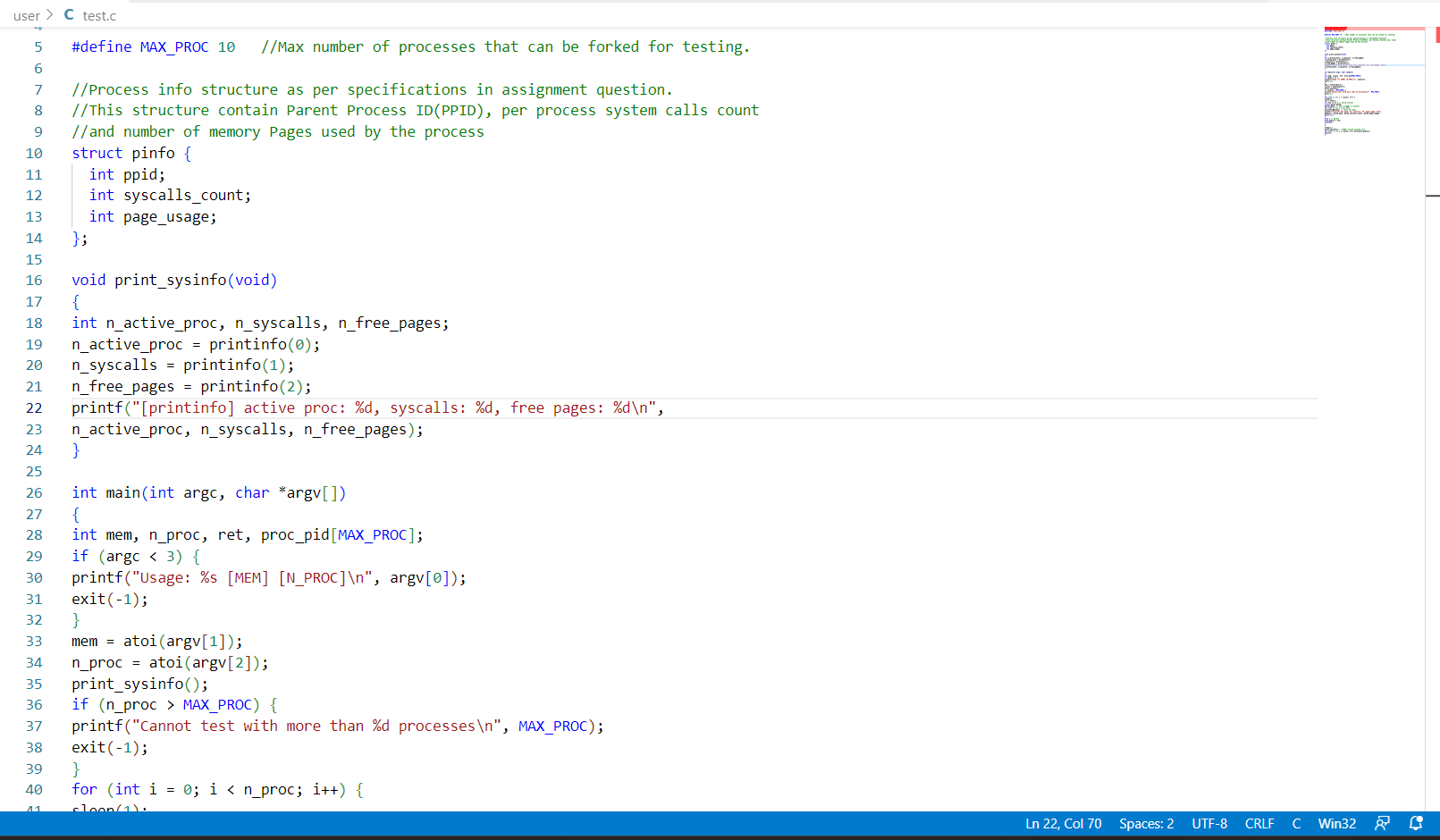
1. **MakeFile**

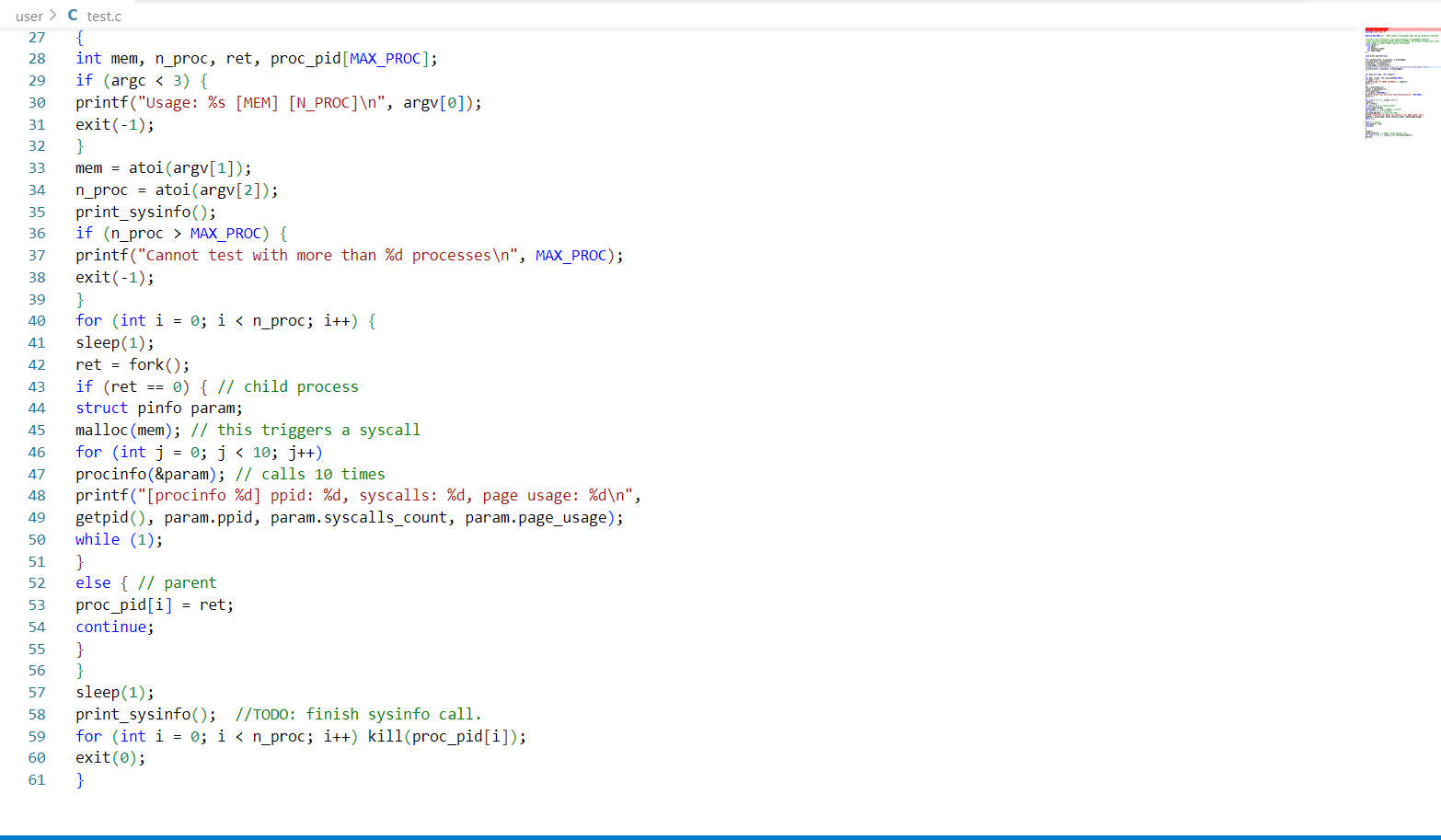
Add test file to make file and change number of CPUs to 1.



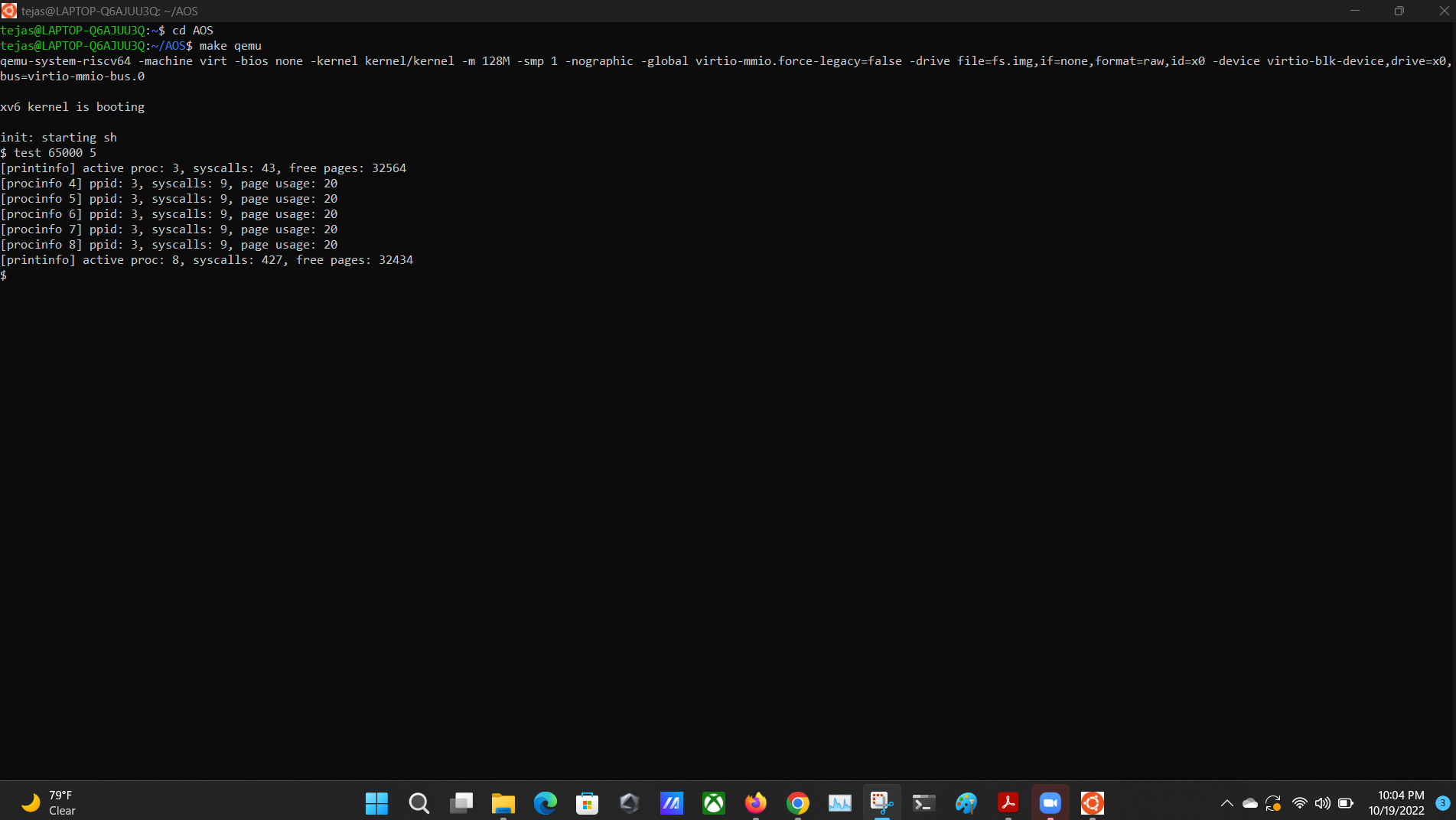
**D. Output:**

**Test.c:** We have used the same test file as mentioned in the Lab1 pdf for testing our implementations.





Output obtained-



**E. Detailed explanation of xv6 code with changes-**

**Part 1:**

Ths XV6 source code from the gitlink contained all the files required for a system call implementation. Our objective was to create a new system call which depending upon the user’s input ( 0,1 or 2) would return the number of active processes in the system , return the number of total system calls since boot up or return the number of free memory pages present in the system respectively. Detailed explanation of each is further below. Overall, when the system call is initiated , it evaluates the argument and passes it to printinfo function, if there is wrong argument , it returns -1.

The valid arguments are-

* **Number of process count:**

When param is 0 , the function proc\_count() is called. The function contains the pointer to proc array which is set UNUSED when initialized . (NPROC - contains the number of processing units available. ). To prevent deadlocks and race conditions and to guarantee that only one CPU can access the data, the process must acquire the lock using the "acquire" keyword. Each lock sustains the acquired state during subsequent calls. For every active process, the counter ct is incremented. Once the function is implemented the lock is released.

* **Total Number of system call :**

When param is 1, the function tot\_sys\_call() is called which in turn calls the getsyscallcount() function. We introduced a global variable system\_call\_count in syscall.c which is intialized to 0 at boot up and gets incremented whenever there is a system call in syscall(). This variable is returned by getsyscallcount() which is later returned by tot\_sys\_call().

* **Free memory pages:**

When param is 2, the function returns the free pages available. The mem\_pages calls a function (which we created) kfreepages() present in kalloc.c. The kinit() function is used when the xv6 kernel is being initialized. This executes the freerange() function, which "frees" all the unused pages that are located above the kernel code and data all the way to the end of physical memory. Free pages are tracked by the xv6 kernel in a single linked list. Firstly, a lock is acquired to avoid deadlocks and racing conditions. Starting from the first node , the linked list is traversed and the counter is incremented as we go till the end. Lock is released once the function is implemented.

This is the general flow of the part 1 of the assignment to implement system call.

**Part 2:**

* **myproc():**

We used this function to fetch the PCB for each process to fetch the PPID, Per Process Sys Call count and the number of Memory Pages used by each process.

* **copyout():**

Since the kernel cannot directly write data to the user space, we used this function to copy the per process data from the kernel space to the user space pinfo structure. We accessed the user space pinfo structure by passing the pointer to user space pinfo structure between the system calls and kernel function calls.

* **argaddr():**

We used this function to retrieve the pointer to the user space pinfo in system call. We passed the pointer as an argument to the system call sys\_procinfo.

* **PPID:**

We get the Parent Process ID from the process PCB.

* **System call count per process:**

We defined a new variable in the process PCB to count the number of system calls executed by each process. We access this variable from PCB by calling the myproc function. We increment this variable in the syscall method.

* **Memory usage per process:**

We find this information also from the process PCB.

* **Flow:**

1. In the test program, we receive the memory to be assigned to each process and the number of processes to be forked.
2. For the number of processes in input, we call the **sys\_procinfo** system call from the user space and pass the address of user space pinfo structure.
3. We receive the address of user space pinfo structure in a pointer variable using the argaddr() function in the procinfo syscall. We execute the kernel call **processinfo** from this sys call and pass the pointer variable received.
4. In the kernel call, we call the myproc() function to fetch the PCB for each process and use the copyout function to copy data from kernel space to the user space pinfo structure.
5. The control returns to the user space test function.

**F. Contributions -**

| **Tejas Milind Deshpande** | * Studied the first part of Lab1 to get more idea about implementing the second part of Lab1. * Setup the private Git repo for collaborating with team members. * Implemented the **processinfo** kernel function with **copyout** functionality. Studied the copyout functionality from file.c. * Implemented the **Parent PID** and **number of Memory Pages** Used part in Lab1 part 2. * Implemented the **sys\_procinfo** system call. * Helped for code integration. * Reviewed code for peers and gave pointers for improvement. * Created the report and video. |
| --- | --- |
| **Satya Sri Nandan Paritala** | * Studied and implemented the test.c file in Lab1 pdf and modified it for our implementation. * Implemented system calls count in part 1 for total system calls and in part 2 for per process system call. * Studied the copyout function from file.c * Declared the system calls and kernel calls in various places. * Reviewed the code for peers. * Created report and video. |
| **Nunna Lakshmi Saranya** | * Defined the System call for first part of the assignment (**Sysinfo)** and implemented for **total number of active processes** and **Total number of free page**s available in the system. * Studied about the page allocation in xv6 and kalloc file in xv6. * Studied the second part of lab1 implementation. * Worked on Sections A,B part 1,C, D part 1 of the lab report. And created the video. * Remaining changes were made along with the teammates. * Reviewed the code, report and video for the assignment |