به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس آزمایشگاه سیستم عامل

پروژه دوم

نام و نام خانوادگی	دانیال سعیدی(810198571) سروش صادقیان(810898048) محمد قره حسنلو(810198461)
تاریخ ارسال گزارش	28/1/1401
ریپو گیتهاب	
آخرین Commit ID	

فهرست گزارش سوالات

سوال 1

ULIB از چهار فایل ulib، printf و umalloc تشکیل شده است.

ulib: از چند تابع تشکیل شده است که در بعضی از آنها فراخوانی سیستمی داریم:

stat: در این تابع از open استفاده شده برای باز کردن فایل و یک خروجی fd است و از close برای بستن فایل باز شده با file descriptor خروجی از open است.

gets: از read برای خواندن استفاده شده و چون پارامتر اول آن صفر است، از همان fd که داخلش است، می خواند

printf: از سه تابع تشکیل شده است که در تابع putc، از سیستم کال write استفاده شده است که در آن کاراکتر c را در fd ورودی مینویسد.

Umalloc: در تابع morecore از سیستم کال sbrk به منظور تغییر دادن data segment استفاده میشود.

سوال 2

علاوه بر فراخوانی سیستمی راه های دیگری برای دسترسی به هسته وجود دارد؛ مانند exception که در لینوکس دسترسی به هسته داریم تا هسته بتواند خطا را از بین ببرد. راه دیگر بر مبنای سوکت است که برنامه هایی که در قسمت برای کاربر نوشته میشود، میتوانند به هسته دسترسی داشته باشند و اطلاعات را به هسته بفرستند ویا دریافت کنند. همچنین برای /sys و /proc و... نیاز به دسترسی هسته داریم که محتوای داخل خود هسته را به برنامه در سطح کاربر میدهیم.

سوال 3

نمیتوان فعال نمود. با فعال کردن بقیه trap ها با سطح DPL_USER که یک سطح دسترسی کاربر است، هر دستوری همزمان میتوانست به kernel دسترسی پیدا کند که مشکل امنیتی ایجاد میشد(چون ممکن است حداقل یکی از این دسترسی ها دارای باگ باشند و در kernel تاثیر بگذارد.)

سوال 4

ss و esp وقتی استک بین دو سطح موجود kernel و user تغییر میکند، عوض میشوند و وقتی این push کردن انجام میشود که تغییر دسترسی انجام شود، چون از استک پشته قبلی

نمیتوان در سطحی که در آن قرار گرفتیم، استفاده کنیم. اگر تغییر سطح رخ ندهد، نیازی به push نیست؛ چون از همان استک قبلی داریم استفاده میکنیم.

سوال 5

argptr: n شماره آرگومان، pp آدرس پوینتر و size اندازه آرگومان، ورودی های آرگومان هستند که مقداری که آرگومان دارد را در pp میریزد.

argint: آدرس یک عدد که با ip مشخص شده را برای گرفتن آرگومانی که به عدد n دادیم، میدهیم.

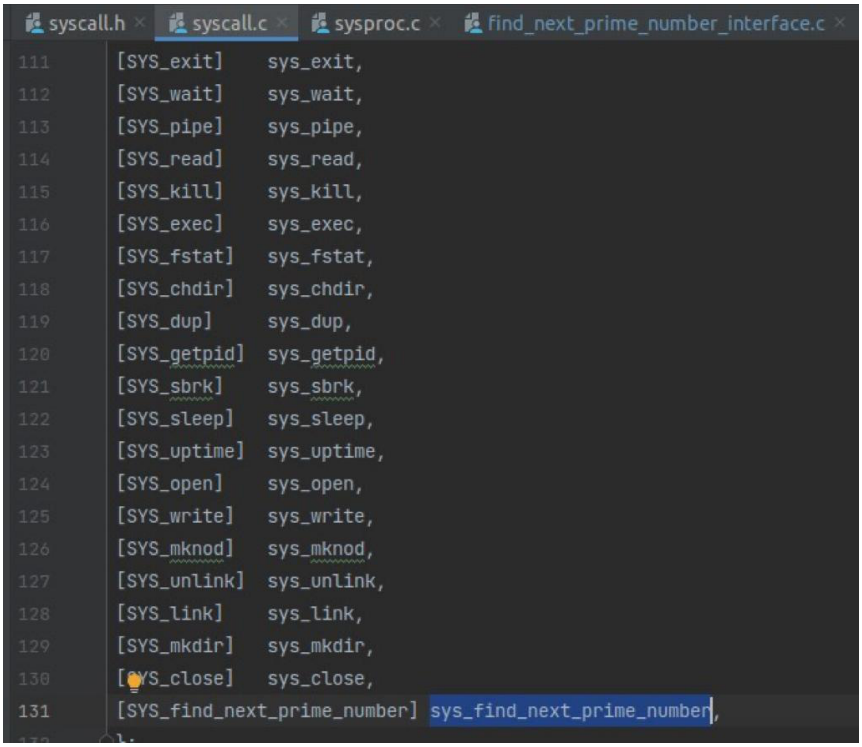
argstr: آرگومان با توجه به شماره آرگومان که ورودی هست، در pp که یک ورودی است ریخته میشود و آن را برمیگرداند.

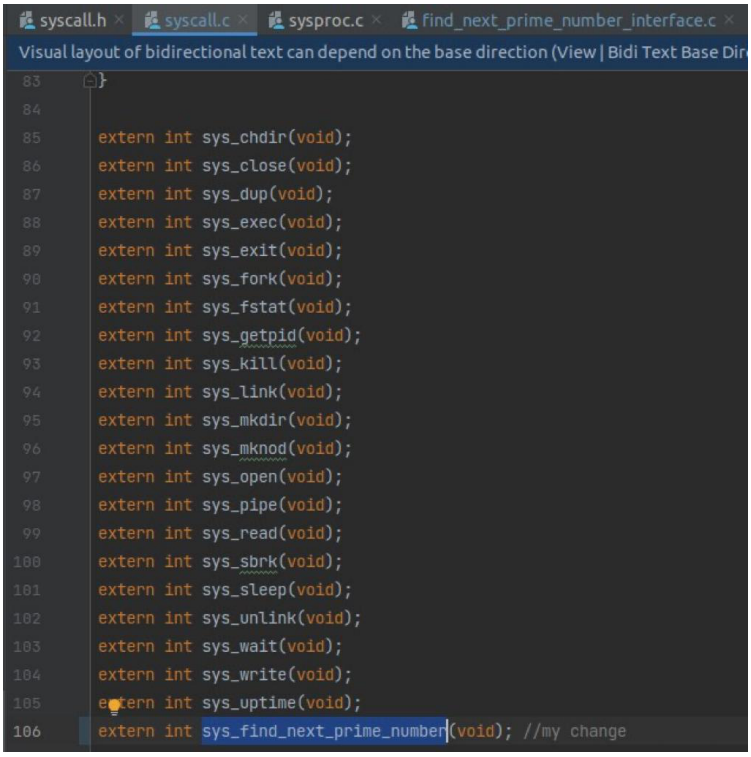
در صورت مشخص نشدن بازه، ممکن بود به آدرس اشتباه و در نتیجه مقدار اشتباه دسترسی پیدا میکردیم که دلیل مشکل امنیتی نیز همین موضوع است.

بخش 0: ارسال آرگومان های فراخوانی های سیستمی

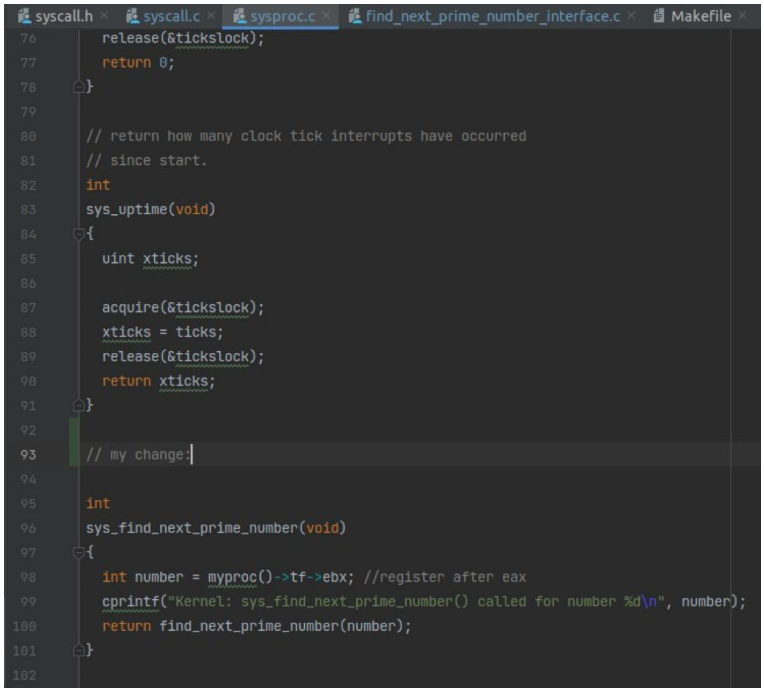
خروجی:

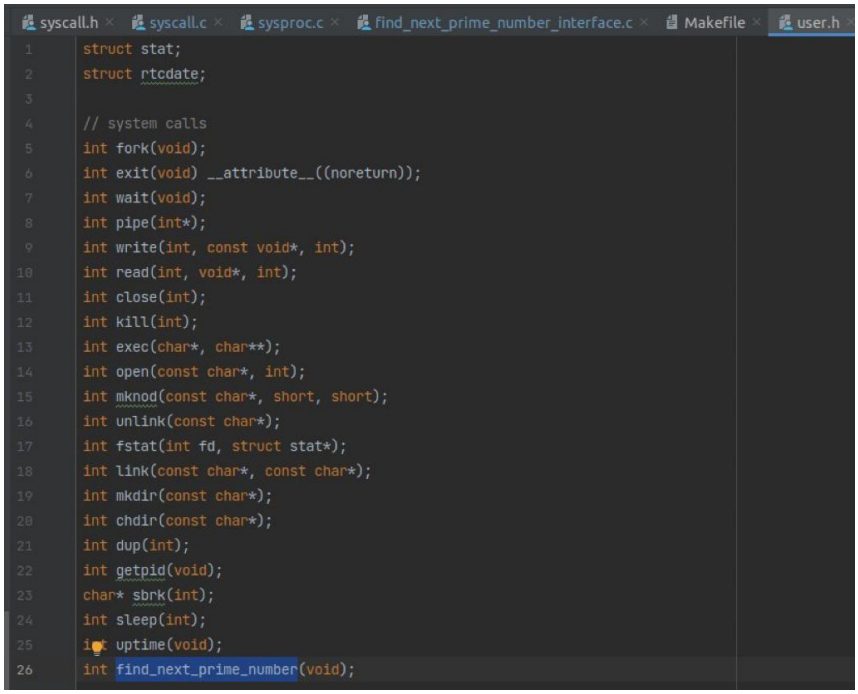
مراحل اضافه کردن فراخوانی سیستمی مربوطه:

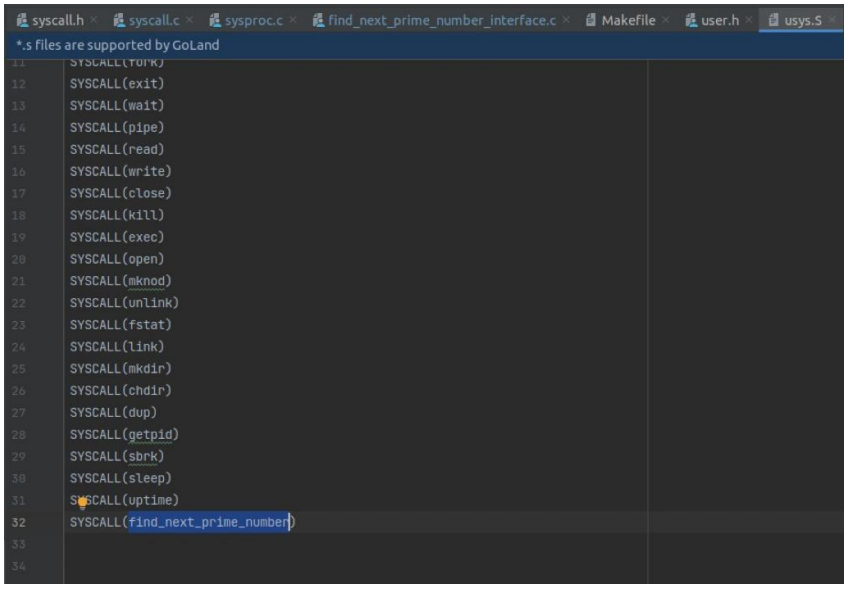




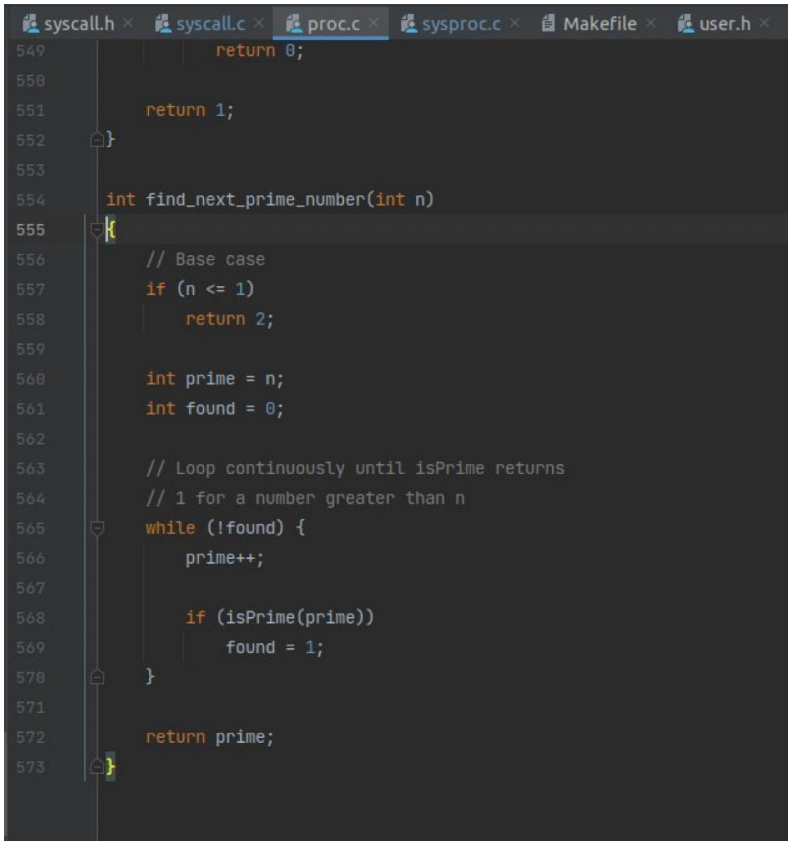
در این قسمت حواسمان باید به این نکته باشد که مقدار رجیستر قبل و بعد از فراخوانی سیستمی تغییر نکند که برای این کار به صورت زیر عمل میکنیم:

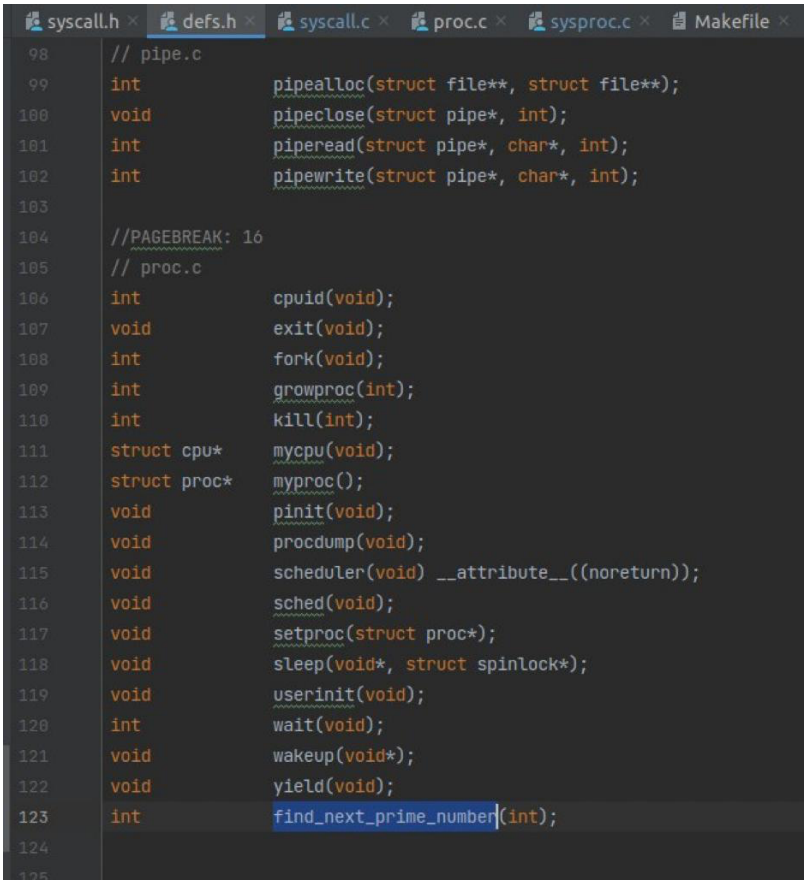






پیاده سازی برنامه:




syscall.h
defs.h
syscall.c
proc.c
sysproc.c
Makefile
// pipe.c
int pipealloc(struct file**, struct file**);
void pipeclose(struct pipe*, int);
int piperead(struct pipe*, char*, int);
int pipewrite(struct pipe*, char*, int);
//PAGEBREAK: 16
// proc.c
int cpuid(void);
void exit(void);
int fork(void);
int growproc(int);
int kill(int);
struct cpu* mycpu(void);
struct proc* myproc();
void pinit(void);
void procdump(void);
void scheduler(void) __attribute__((noreturn));
void sched(void);
void setproc(struct proc*);
void sleep(void*, struct spinlock*);
void userinit(void);
int wait(void);
void wakeup(void*);
void yield(void);
int find_next_prime_number(int);

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_find_next_prime_number 22

```

syscall.c | find_next_prime_number_interface.c | getpid_gdb.asm | getpid_gdb.c

*.c files are supported by CLion

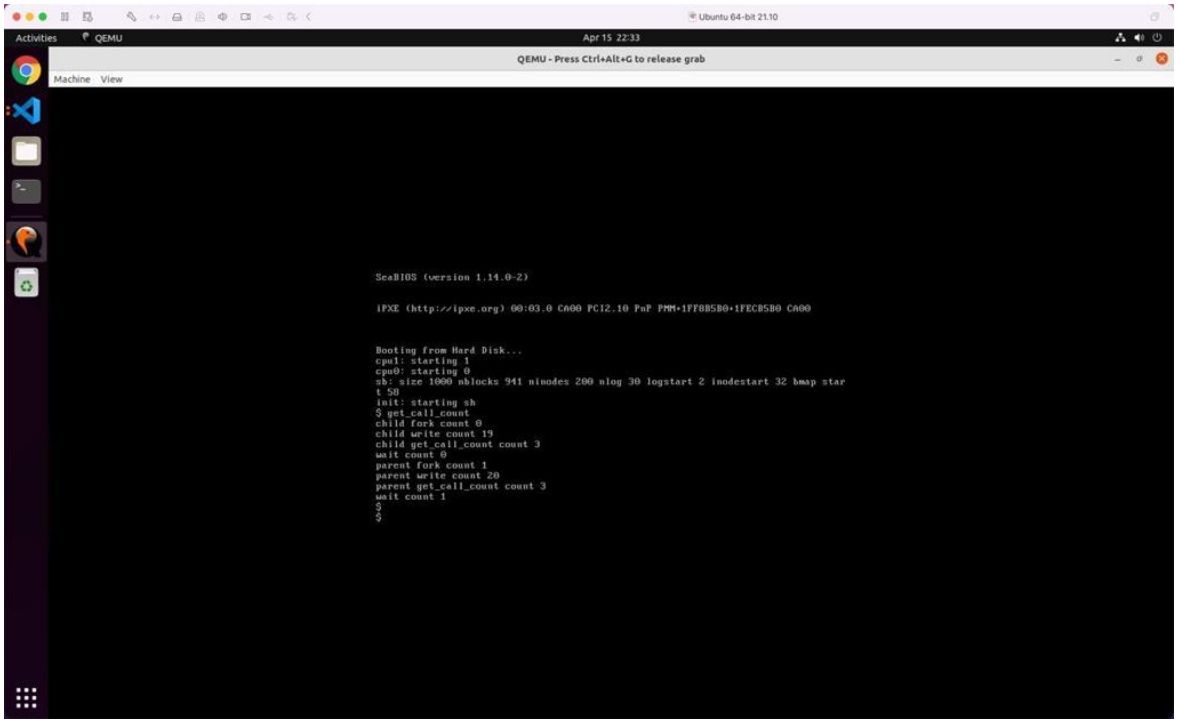
Visual layout of bidirectional text can depend on the base direction (View | Bidi Text Base Direction)

```c
#include "types.h"
#include "stat.h"
#include "user.h"

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

        int saved_ebx, number = atoi(argv[1]);
        // we can add assembly instructions using asm
        asm volatile(
            "movl %%ebx, %0;"
            "movl %1, %%ebx;"
            : "=r" (saved_ebx)
            : "r"(number)
        );

        printf(1, "The next prime number is:  %d\n", find_next_prime_number());
        asm("movl %0, %%ebx" : : "r"(saved_ebx));
        exit();
}
```

بخش 1: پیاده سازی فراخوانی سیستمی شمارش فراخوانی های سیستمی

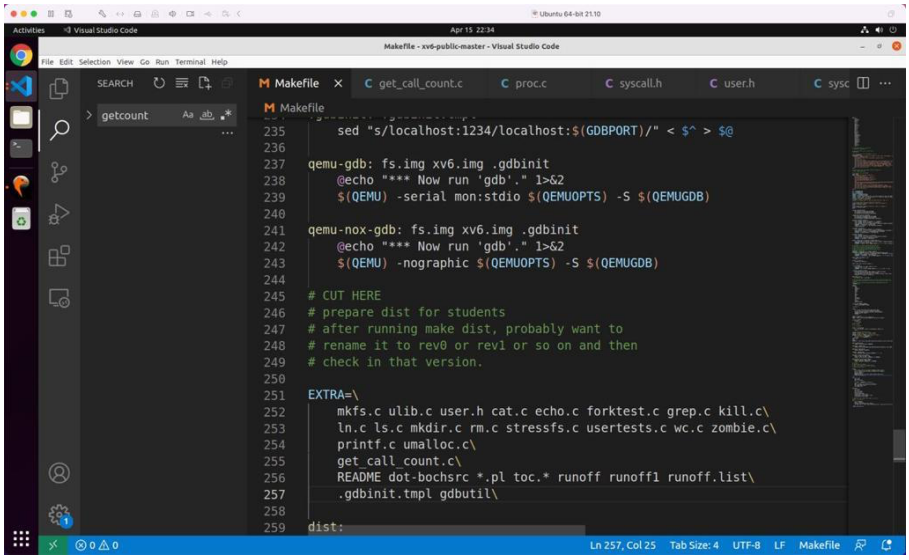
مراحل به صورت زیر انجام شده است:

خروجی:

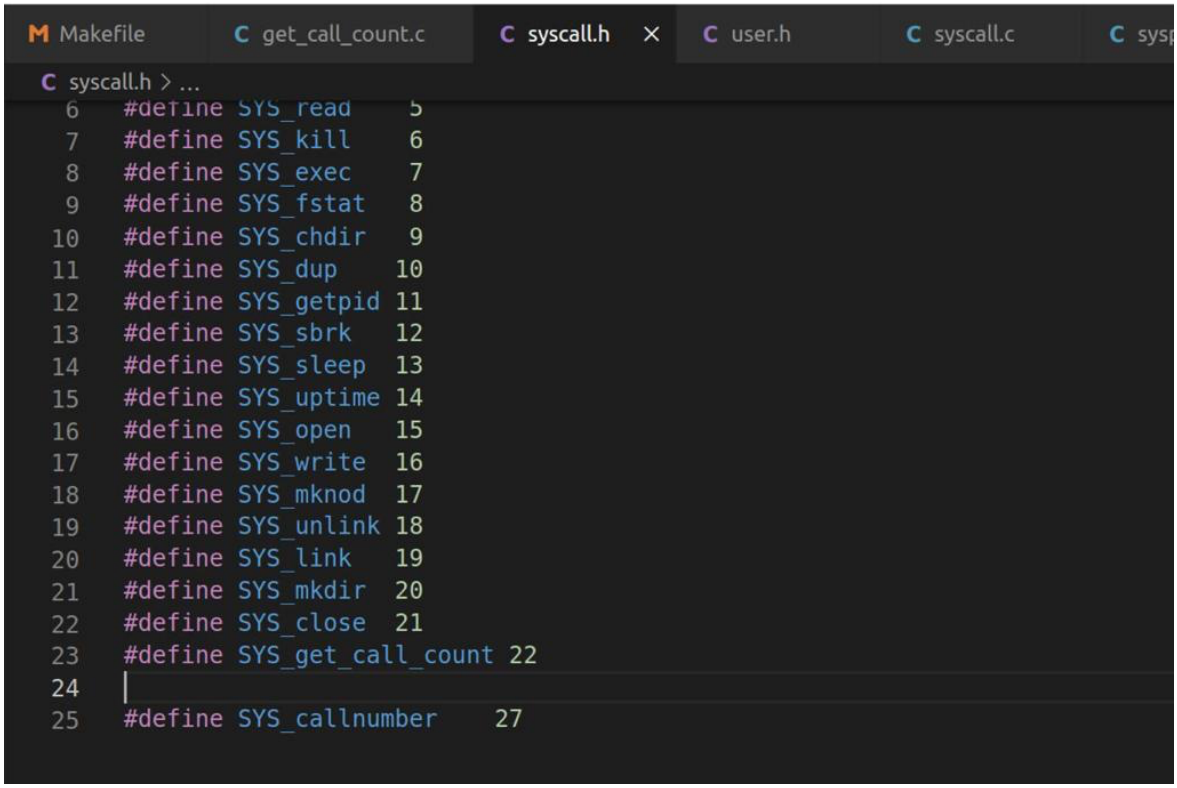


در makefile تغییرات را به صورت زیر اعمال میکنیم:

```
167
168 UPROGS=\
169     _cat\
170     _echo\
171     _forktest\
172     _grep\
173     _init\
174     _kill\
175     _ln\
176     _ls\
177     _mkdir\
178     _rm\
179     _sh\
180     _stressfs\
181     _usertests\
182     _wc\
183     _zombie\
184     _get_call_count\
185
```



برای اضافه کردن فراخوانی سیستمی مورد نظر، قدم های زیر را برمیداریم:



برنامه برای تست فراخوانی سیستمی اضافه شده:

Makefile | get_call_count.c | proc.c | syscall.h | user.h

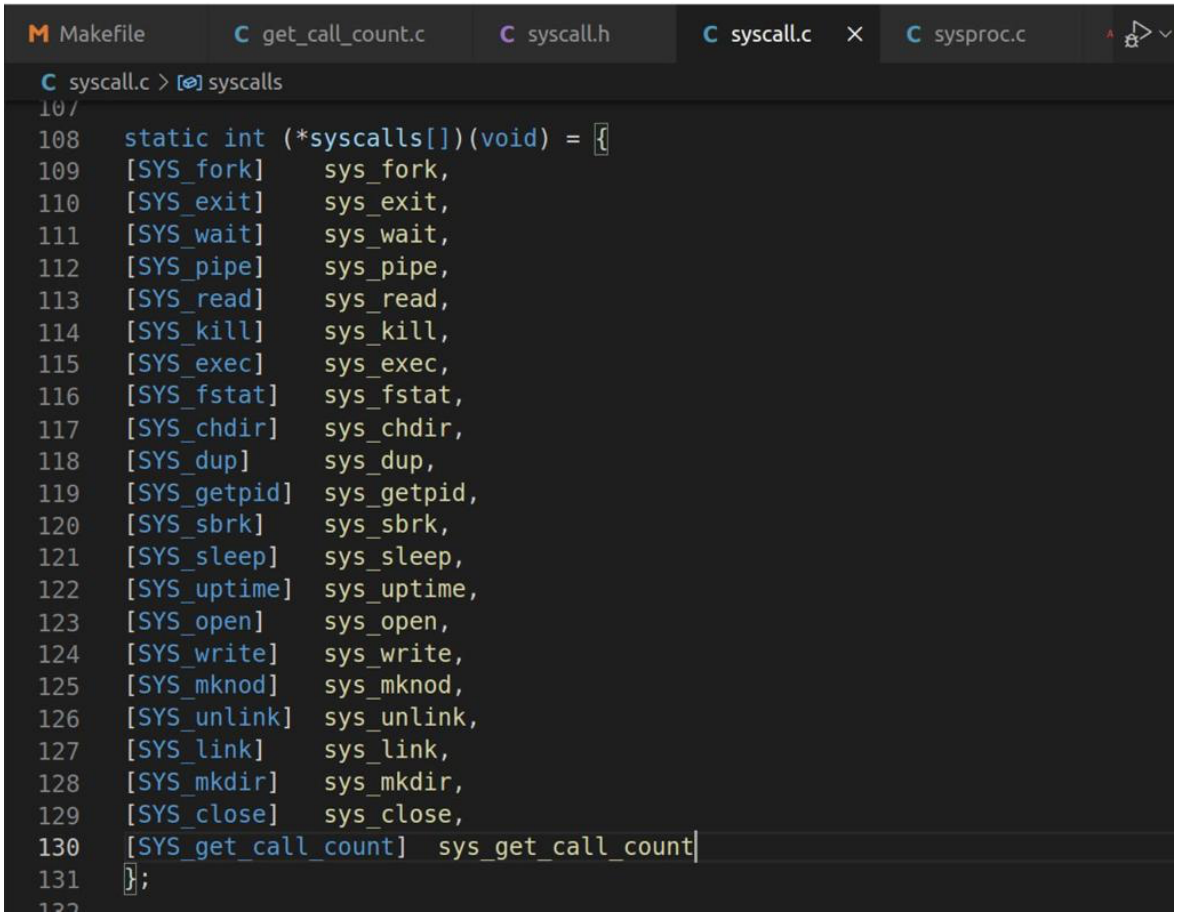
get_call_count.c > main(int, char * [])

```c
#include "types.h"
#include "user.h"
#include "syscall.h"

int
main(int argc, char *argv[])
{
  if (fork() == 0) {
    printf(1, "child fork count %d\n", get_call_count(SYS_fork));
    printf(1, "child write count %d\n", get_call_count(SYS_write));
  printf(1, "child get_call_count count %d\n", get_call_count(SYS_get_call
  } else {
    wait();
    printf(1, "parent fork count %d\n", get_call_count(SYS_fork));
    printf(1, "parent write count %d\n", get_call_count(SYS_write));
  printf(1, "parent get_call_count count %d\n", get_call_count(SYS_get_cal
  }
  printf(1, "wait count %d\n", get_call_count(SYS_wait));
  exit();
}

```

Makefile | get_call_count.c | syscall.h | user.h | syscall.c | sysp

user.h > get_call_count(int)

```c
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int get_call_count(int sys_call);
```


Makefile
get_call_count.c
syscall.h
syscall.c
sysproc.c
syscall.c > syscalls
107
108 static int (*syscalls[])(void) = {
109 [SYS_fork] sys_fork,
110 [SYS_exit] sys_exit,
111 [SYS_wait] sys_wait,
112 [SYS_pipe] sys_pipe,
113 [SYS_read] sys_read,
114 [SYS_kill] sys_kill,
115 [SYS_exec] sys_exec,
116 [SYS_fstat] sys_fstat,
117 [SYS_chdir] sys_chdir,
118 [SYS_dup] sys_dup,
119 [SYS_getpid] sys_getpid,
120 [SYS_sbrk] sys_sbrk,
121 [SYS_sleep] sys_sleep,
122 [SYS_uptime] sys_uptime,
123 [SYS_open] sys_open,
124 [SYS_write] sys_write,
125 [SYS_mknod] sys_mknod,
126 [SYS_unlink] sys_unlink,
127 [SYS_link] sys_link,
128 [SYS_mkdir] sys_mkdir,
129 [SYS_close] sys_close,
130 [SYS_get_call_count] sys_get_call_count
131 };

Makefile | get_call_count.c | syscall.h | syscall.c | sysproc.c

syscall.c > syscalls

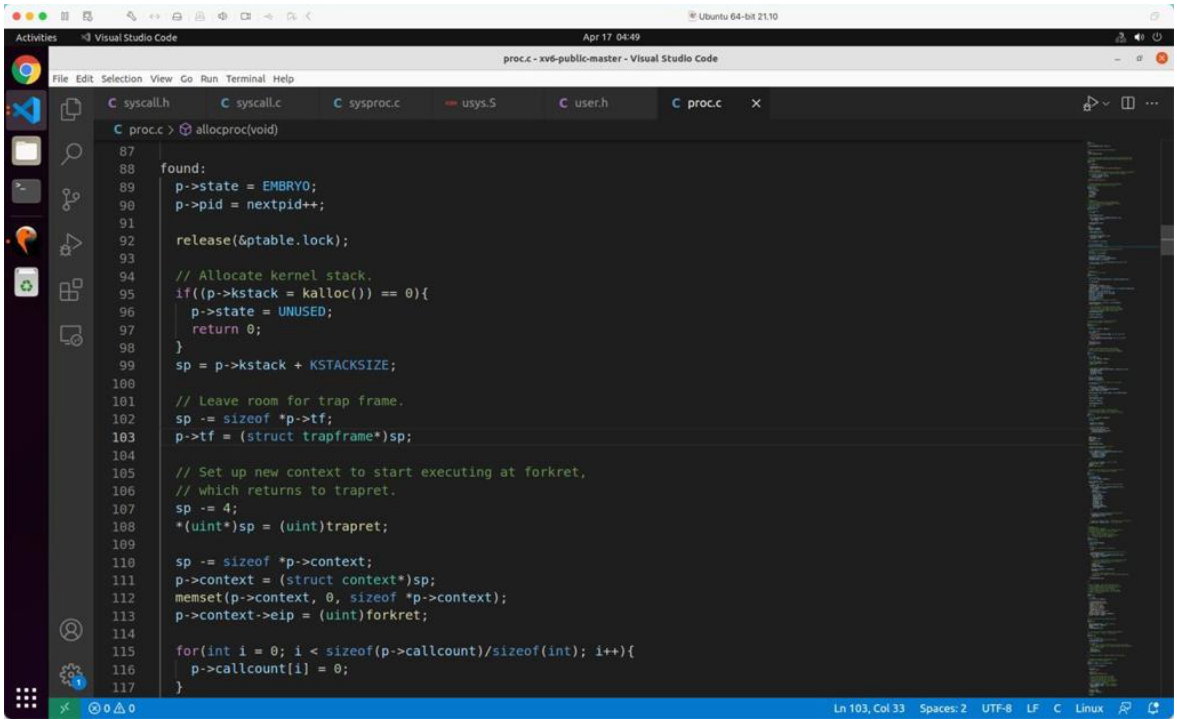
```
130   [SYS_get_call_count]  sys_get_call_count
131   };
132
133   void
134   syscall(void)
135   {
136     int num;
137     struct proc *curproc = myproc();
138
139     num = curproc->tf->eax;
140     curproc->callcount[num - 1] = curproc->callcount[num - 1] + 1;
141     // cprintf("Call Count: %d\n",
142     //           curproc->callcount[num - 1]);
143     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
144
145       curproc->tf->eax = syscalls[num]();
146     } else {
147       cprintf("%d %s: unknown sys call %d\n",
148               curproc->pid, curproc->name, num);
149       curproc->tf->eax = -1;
150     }
151   }
152
```

file | get_call_count.c | syscall.h | syscall.c | sysproc.c | usys.S

usys.S

```
20  SYSCALL(open)
21  SYSCALL(mknod)
22  SYSCALL(unlink)
23  SYSCALL(fstat)
24  SYSCALL(link)
25  SYSCALL(mkdir)
26  SYSCALL(chdir)
27  SYSCALL(dup)
28  SYSCALL(getpid)
29  SYSCALL(sbrk)
30  SYSCALL(sleep)
31  SYSCALL(uptime)
32  SYSCALL(get_call_count)
```

خط 115 و 116 به proc.c برای مقداردهی اولیه اضافه شده است.



Makefile | get_call_count.c | syscall.h | syscall.c | sysproc.c

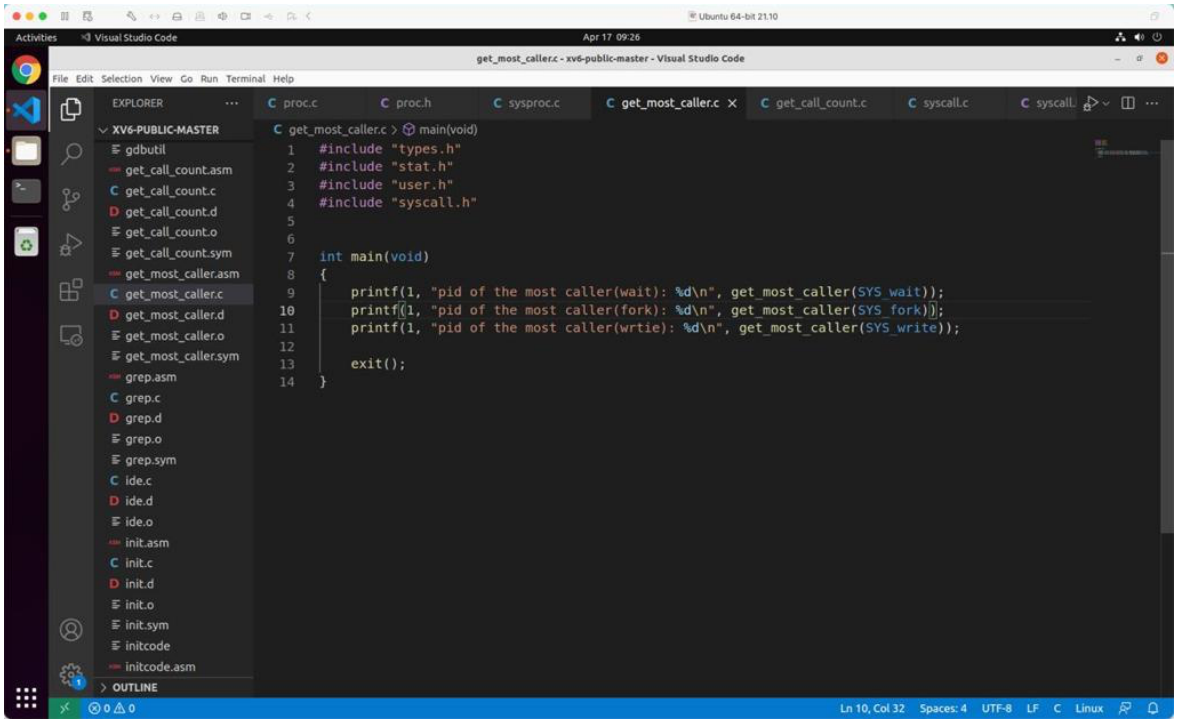
sysproc.c > ...

```c
#include "types.h"
#include "x86.h"
#include "defs.h"
#include "date.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"

int sys_get_call_count(void){
  int n;
  if(argint(0, &n) < 0)
    return -1;
  return(myproc()->callcount[n-1]);
}

int
sys_fork(void)
{
  return fork();
}

int
sys_exit(void)
{
```

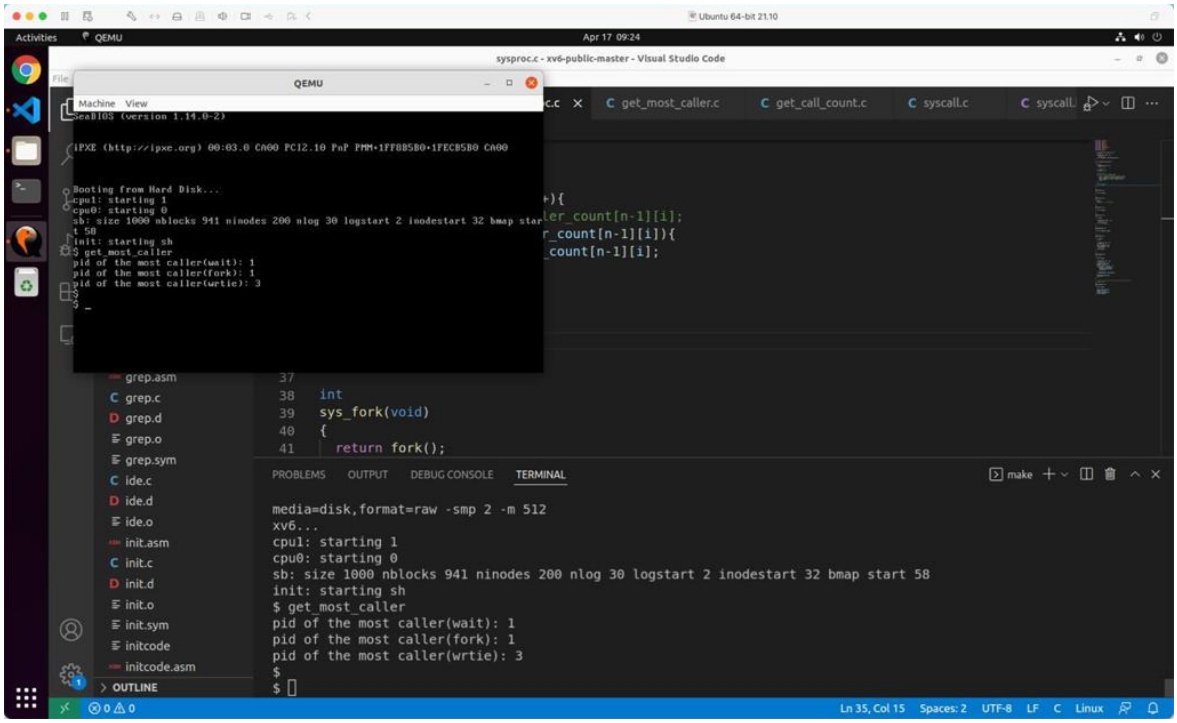
بخش 2: پیاده سازی فراخوانی سیستمی پرمصرف ترین پردازه

توجیه: process یک چون اومده چند تا پراسس دیگه درست کرده خودش واسه همین fork بیشترین دفعات توسط خود پراسس ۱ انجام شده

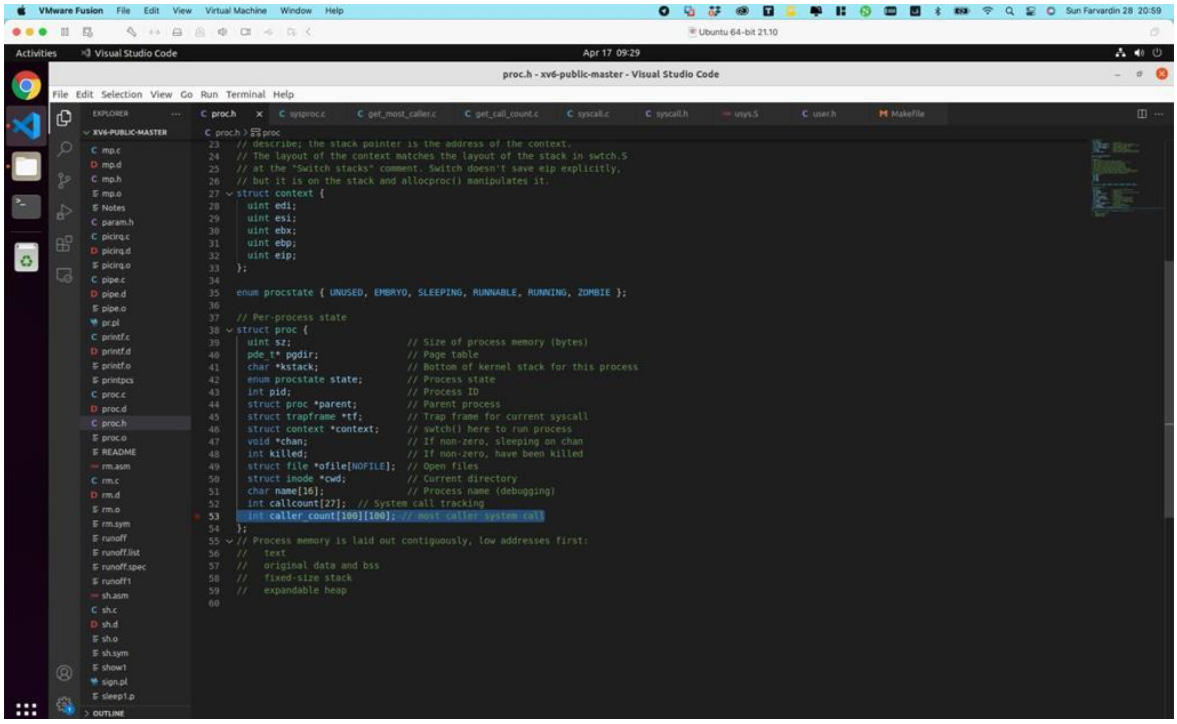
در عکس زیر برنامه تست نوشته شده است:



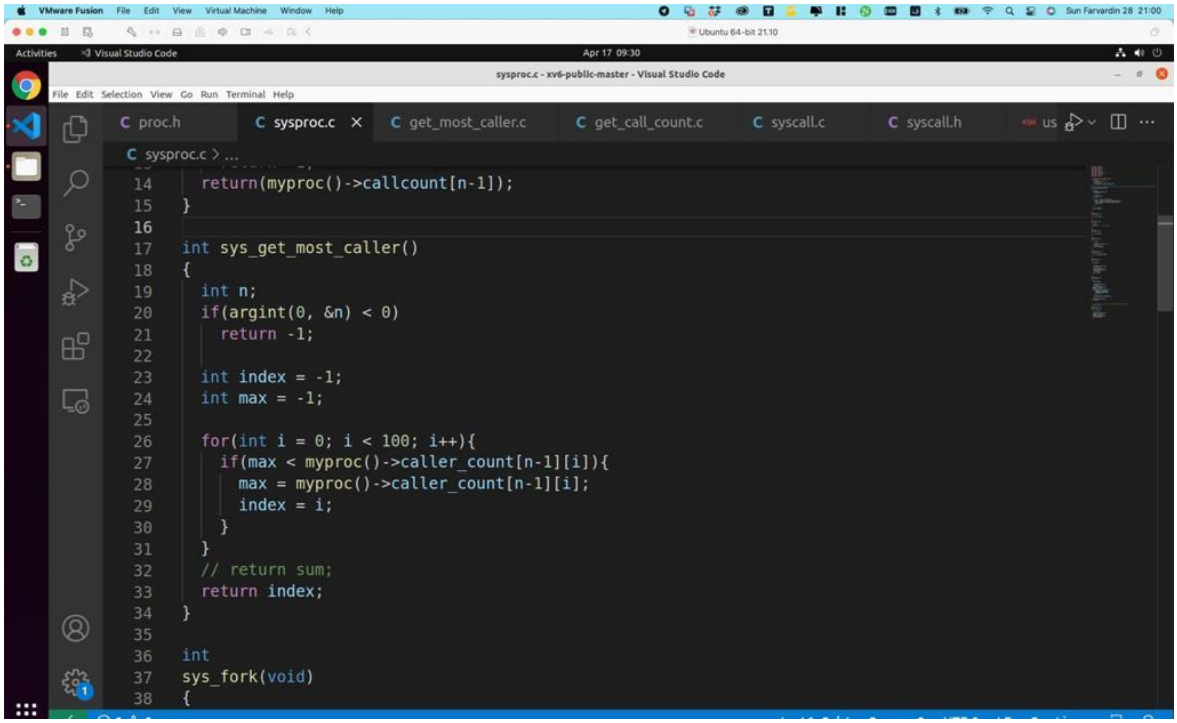
خروجی:

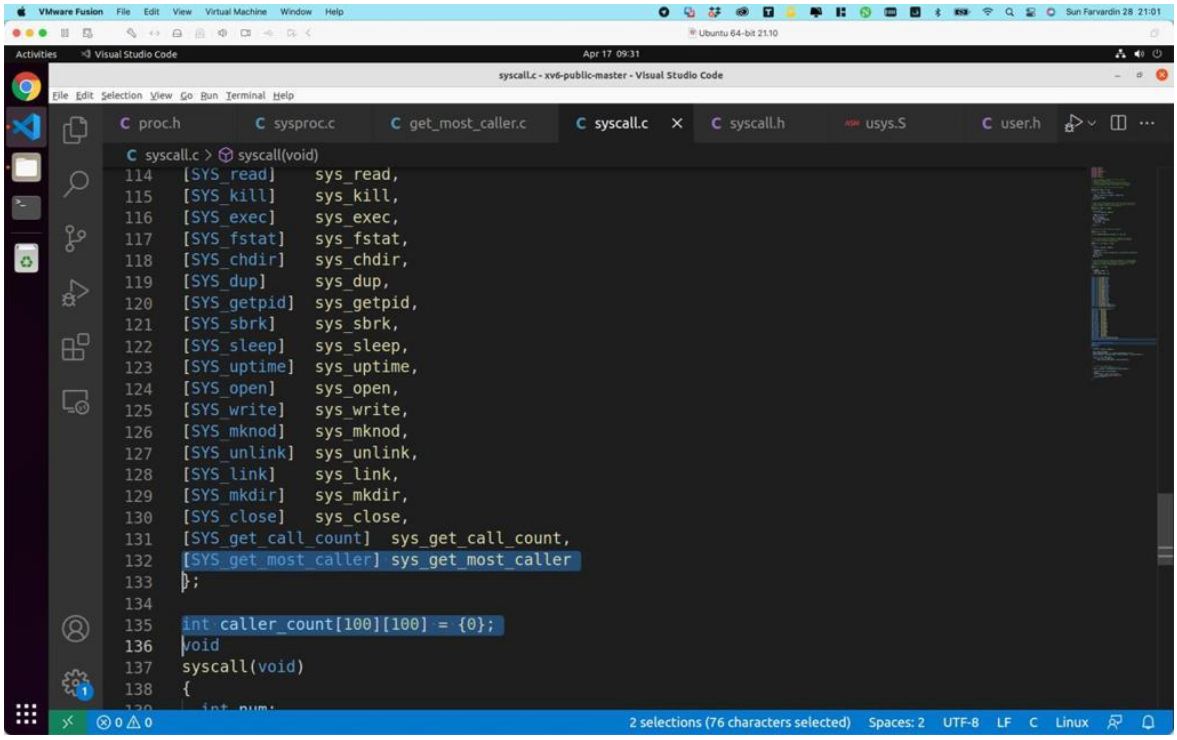


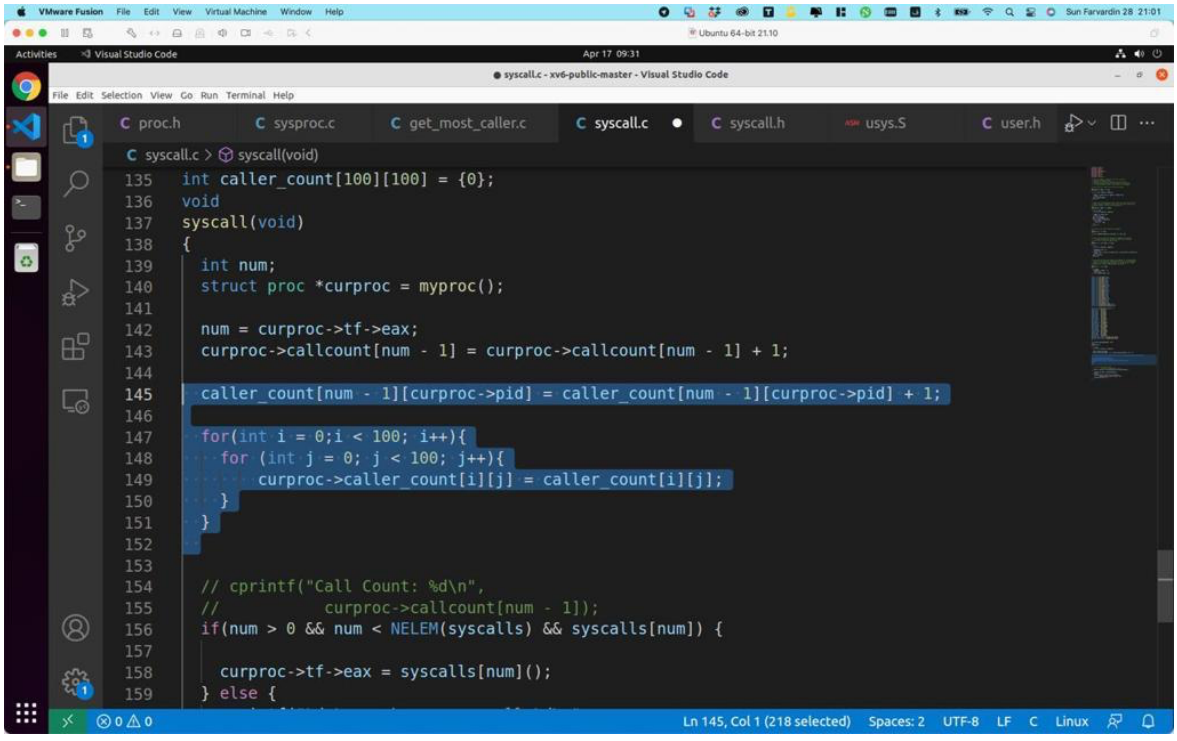
برای اضافه کردن فراخوانی سیستمی این قسمت به صورت زیر عمل میکنیم:

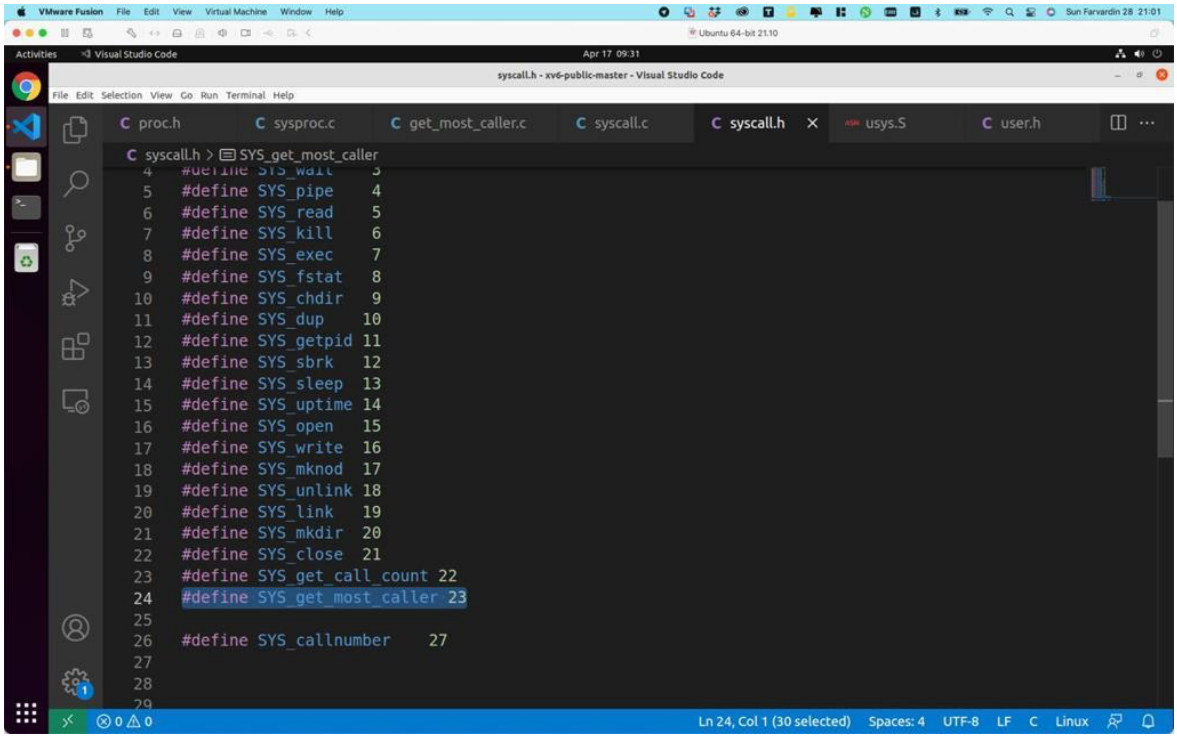


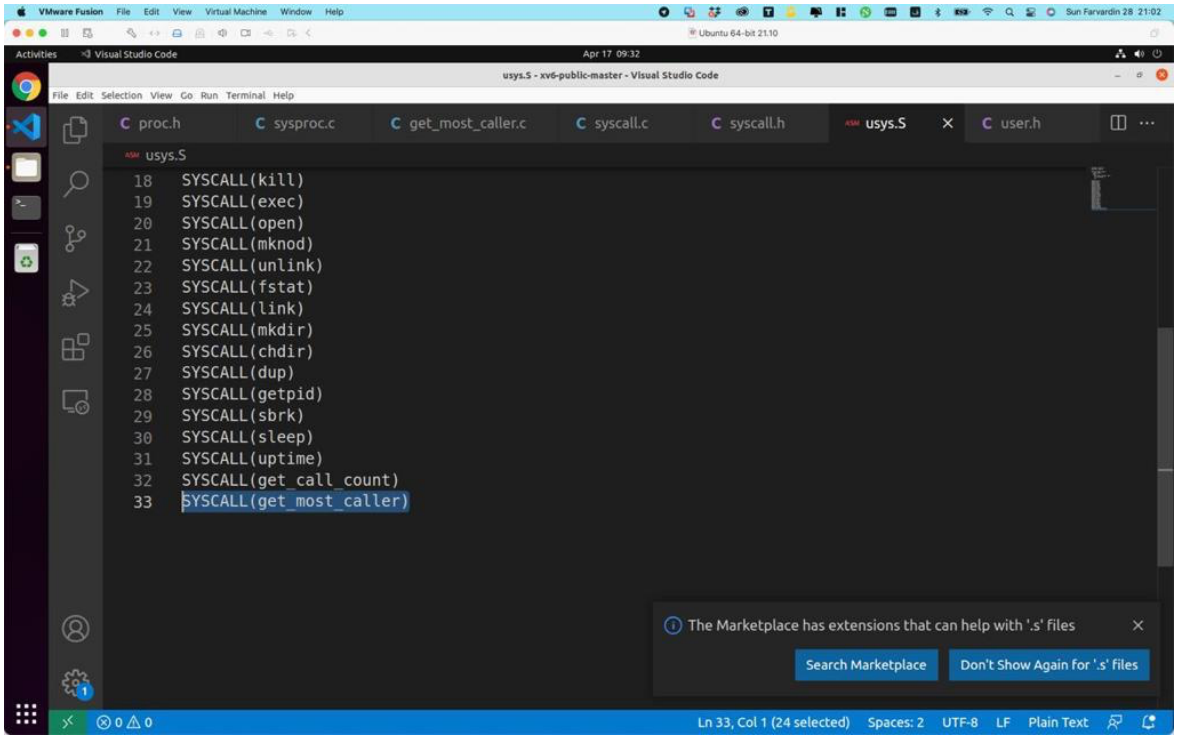
شیوه پیاده سازی:

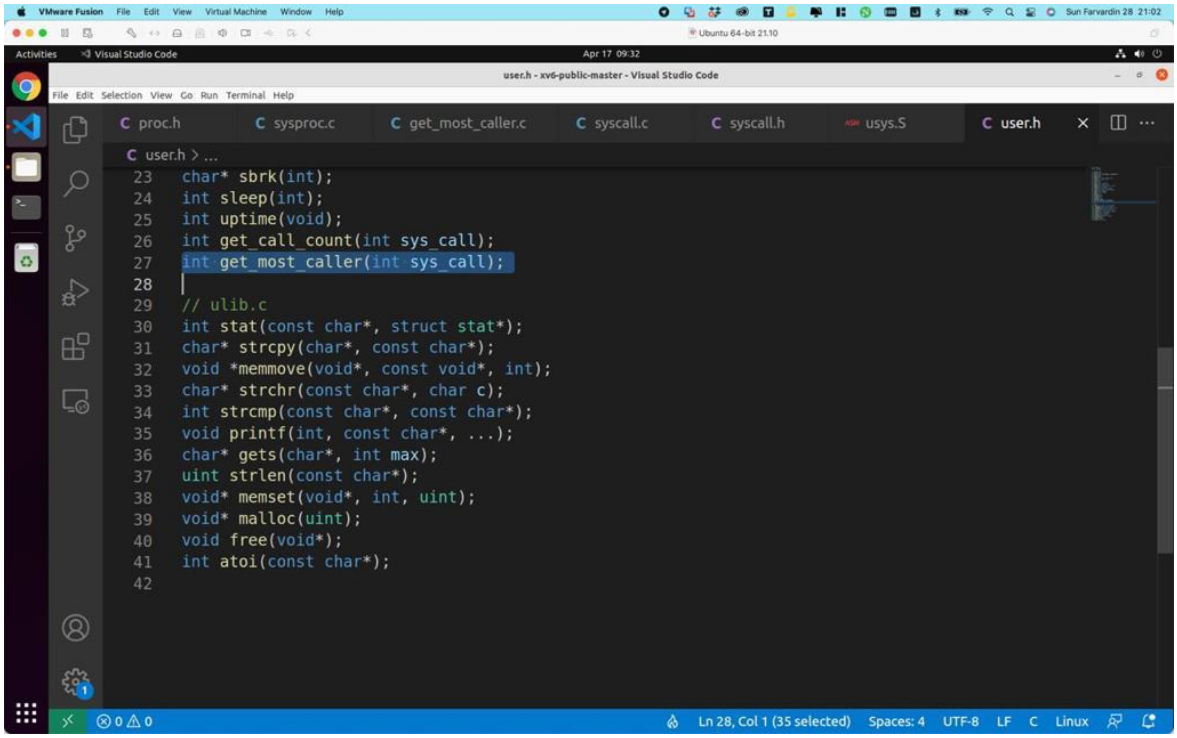



syscall.c - xv6-public-master - Visual Studio Code
proc.h
sysproc.c
get_most_caller.c
syscall.c
syscall.h
usys.S
user.h
syscall.c > syscall(void)
[SYS_read] sys_read,
[SYS_kill] sys_kill,
[SYS_exec] sys_exec,
[SYS_fstat] sys_fstat,
[SYS_chdir] sys_chdir,
[SYS_dup] sys_dup,
[SYS_getpid] sys_getpid,
[SYS_sbrk] sys_sbrk,
[SYS_sleep] sys_sleep,
[SYS_uptime] sys_uptime,
[SYS_open] sys_open,
[SYS_write] sys_write,
[SYS_mknod] sys_mknod,
[SYS_unlink] sys_unlink,
[SYS_link] sys_link,
[SYS_mkdir] sys_mkdir,
[SYS_close] sys_close,
[SYS_get_call_count] sys_get_call_count,
[SYS_get_most_caller] sys_get_most_caller
};
int caller_count[100][100] = {0};
void
syscall(void)
{
2 selections (76 characters selected)
Spaces: 2
UTF-8
LF
C
Linux

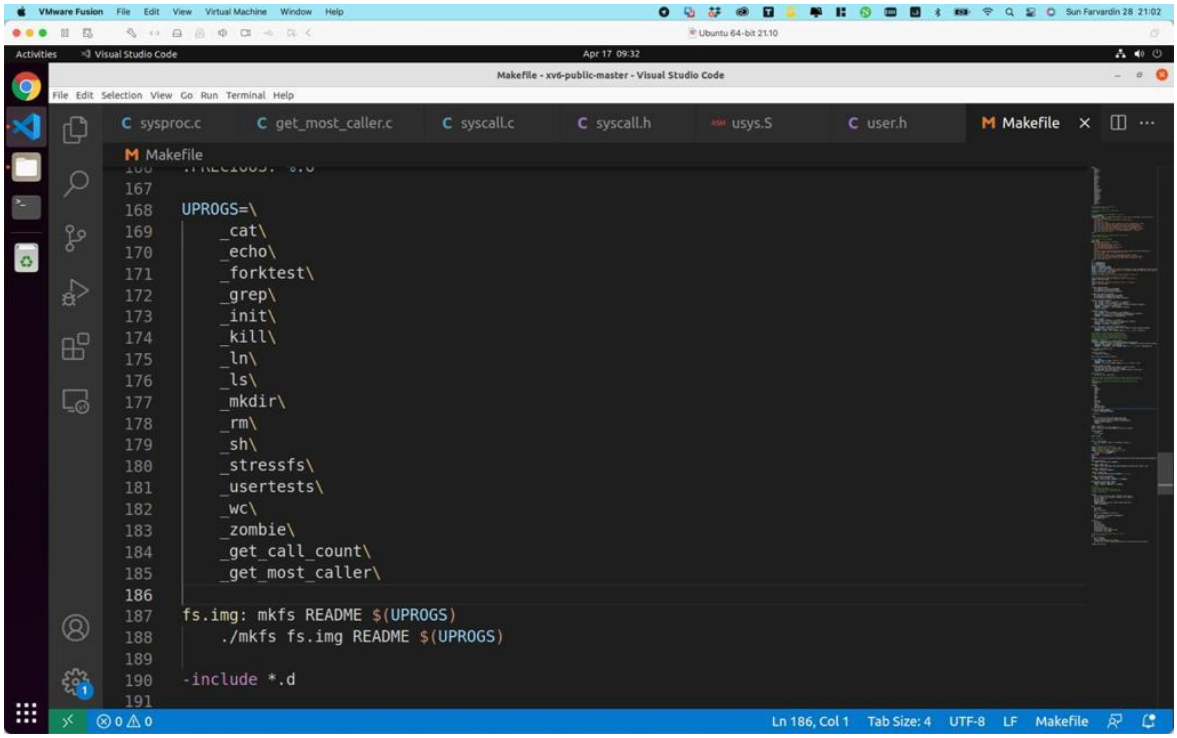

syscall.c - xv6-public-master - Visual Studio Code
proc.h
sysproc.c
get_most_caller.c
syscall.c
syscall.h
usys.S
user.h
syscall.c > syscall(void)
int caller_count[100][100] = {0};
void
syscall(void)
{
int num;
struct proc *curproc = myproc();
num = curproc->tf->eax;
curproc->callcount[num - 1] = curproc->callcount[num - 1] + 1;
caller_count[num - 1][curproc->pid] = caller_count[num - 1][curproc->pid] + 1;
for(int i = 0;i < 100; i++){
for (int j = 0; j < 100; j++){
curproc->caller_count[i][j] = caller_count[i][j];
}
}
// cprintf("Call Count: %d\n",
// curproc->callcount[num - 1]);
if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
curproc->tf->eax = syscalls[num]();
} else {
Ln 145, Col 1 (218 selected)
Spaces: 2
UTF-8
LF
C
Linux


syscall.h - xv6-public-master - Visual Studio Code
proc.h
sysproc.c
get_most_caller.c
syscall.c
syscall.h
usys.S
user.h
syscall.h > SYS_get_most_caller
#define SYS_pipe 4
#define SYS_read 5
#define SYS_kill 6
#define SYS_exec 7
#define SYS_fstat 8
#define SYS_chdir 9
#define SYS_dup 10
#define SYS_getpid 11
#define SYS_sbrk 12
#define SYS_sleep 13
#define SYS_uptime 14
#define SYS_open 15
#define SYS_write 16
#define SYS_mknod 17
#define SYS_unlink 18
#define SYS_link 19
#define SYS_mkdir 20
#define SYS_close 21
#define SYS_get_call_count 22
#define SYS_get_most_caller 23
#define SYS_callnumber 27
Ln 24, Col 1 (30 selected) Spaces: 4 UTF-8 LF C Linux


usys.S - xv6-public-master - Visual Studio Code
proc.h
sysproc.c
get_most_caller.c
syscall.c
syscall.h
usys.S
user.h
usys.S
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(get_call_count)
SYSCALL(get_most_caller)
The Marketplace has extensions that can help with '.s' files
Search Marketplace
Don't Show Again for '.s' files
Ln 33, Col 1 (24 selected) Spaces: 2 UTF-8 LF Plain Text



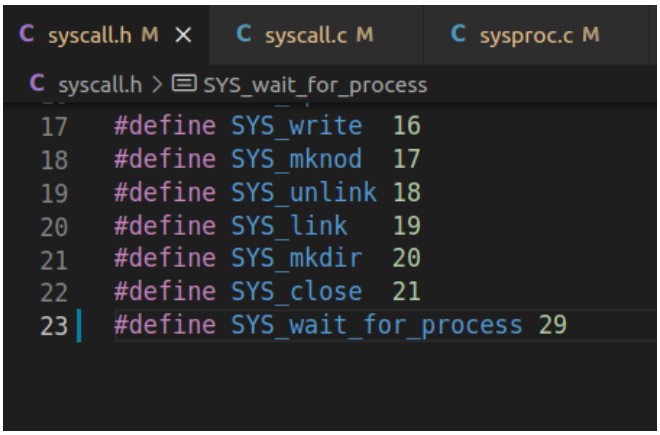
تغییرات makefile برای فراخوانی سیستمی این قسمت به صورت زیر است:



```
252  EXTRA=\
253      mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
254      ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
255      printf.c umalloc.c\
256      get_call_count.c\
257      get_most_caller.c\
258      README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
259      .gdbinit.tmpl gdbutil\
```

بخش 3: پیاده سازی فراخوانی سیستمی صبر برای پردازه های دیگر

مراحل به صورت زیر انجام شده است:



syscall.h M | syscall.c M | sysproc.c M | defs.h M

syscall.c > syscalls

```
104  extern int sys_write(void);
105  extern int sys_uptime(void);
106  extern int sys_wait_for_process(void);
107
108  static int (*syscalls[])(void) = {
109  [SYS_fork]    sys_fork,
110  [SYS_exit]    sys_exit,
111  [SYS_wait]    sys_wait,
112  [SYS_pipe]    sys_pipe,
113  [SYS_read]    sys_read,
114  [SYS_kill]    sys_kill,
115  [SYS_exec]    sys_exec,
116  [SYS_fstat]   sys_fstat,
117  [SYS_chdir]   sys_chdir,
118  [SYS_dup]     sys_dup,
119  [SYS_getpid]  sys_getpid,
120  [SYS_sbrk]    sys_sbrk,
121  [SYS_sleep]   sys_sleep,
122  [SYS_uptime]  sys_uptime,
123  [SYS_open]    sys_open,
124  [SYS_write]   sys_write,
125  [SYS_mknod]   sys_mknod,
126  [SYS_unlink]  sys_unlink,
127  [SYS_link]    sys_link,
128  [SYS_mkdir]   sys_mkdir,
129  [SYS_close]   sys_close,
130  [SYS_wait_for_process] sys_wait_for_process,
131  };
```

syscall.h M | syscall.c M | sysproc.c M | C

sysproc.c > sys_wait_for_process(void)

```
83   sys_uptime(void)
84   {
85     uint xticks;
86
87     acquire(&tickslock);
88     xticks = ticks;
89     release(&tickslock);
90     return xticks;
91   }
92
93   int
94   sys_wait_for_process(void)
95   {
96     int pid;
97     if(argint(0, &pid)<0){
98       return -1;
99     }
100    else{
101      return wait_for_process(pid);
102    }
103
```

syscall.h M | syscall.c M | sysproc.c M | defs.h M | proc.c M | proc.h M | usys.S M | us

usys.S

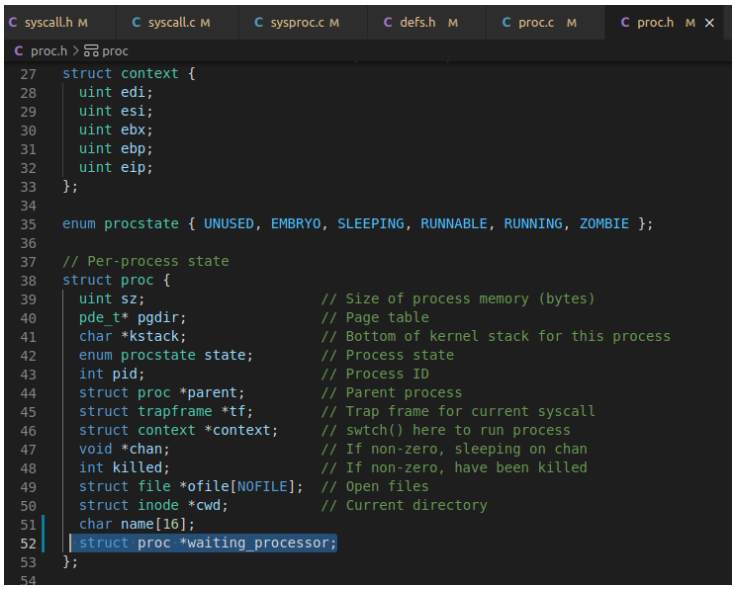
```
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(wait_for_process)
```

syscall.h M | syscall.c M | sysproc.c M | defs.h M | proc.c M

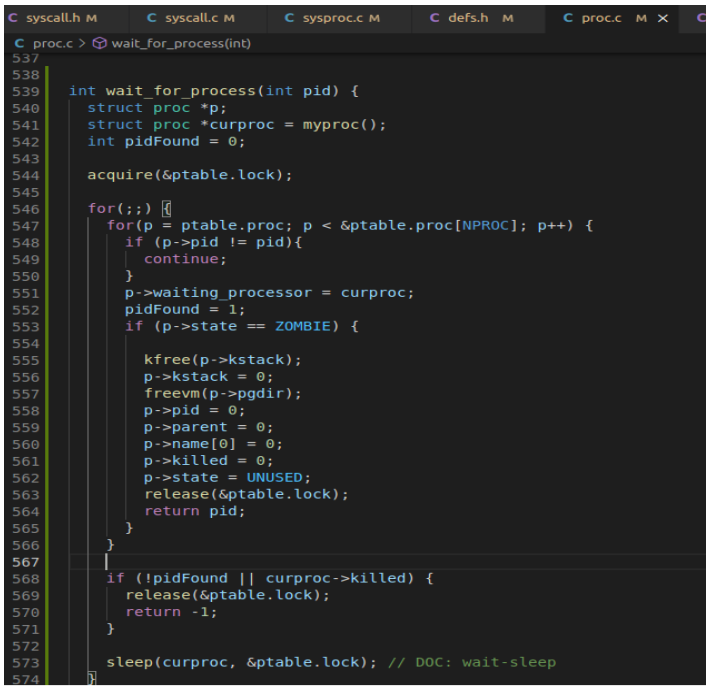
defs.h > wait_for_process(int)

```
// pipe.c
int             pipealloc(struct file**, struct file**);
void            pipeclose(struct pipe*, int);
int             piperead(struct pipe*, char*, int);
int             pipewrite(struct pipe*, char*, int);

//PAGEBREAK: 16
// proc.c
int             cpuid(void);
void            exit(void);
int             fork(void);
int             growproc(int);
int             kill(int);
struct cpu*     mycpu(void);
struct proc*    myproc();
void            pinit(void);
void            procdump(void);
void            scheduler(void) __attribute__((noreturn));
void            sched(void);
void            setproc(struct proc*);
void            sleep(void*, struct spinlock*);
void            userinit(void);
int             wait(void);
int             wait_for_process(int);
void            wakeup(void*);
void            yield(void);
```

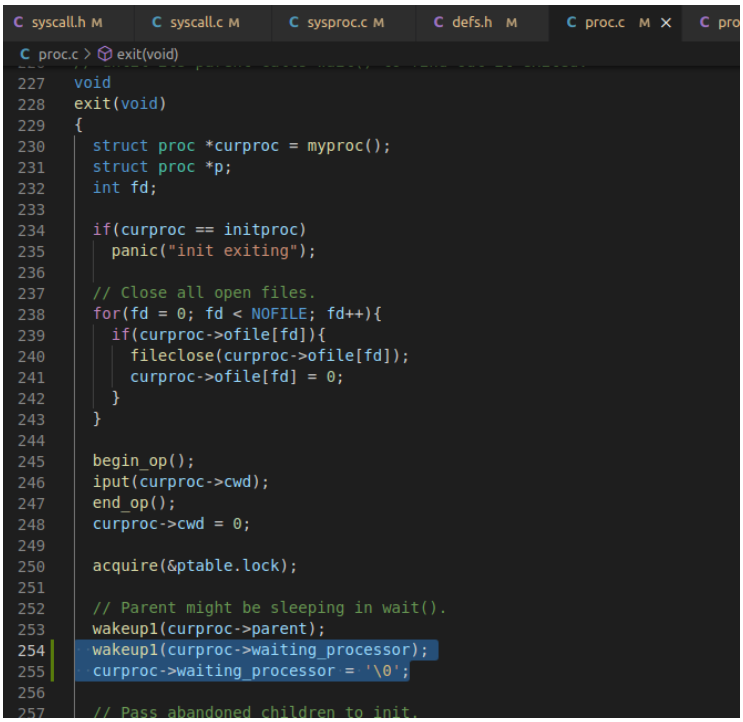
در استرکچر proc یک متود به نام waiting_processor اضافه میشود تا در ادامه از آن استفاده کنیم:



در این قسمت، فراخوانی سیستمی wait_for_process تعریف میشود که از فراخوانی سیستمی wait الگو گرفته شده، با این تفاوت که در حلقه ما به جای parent pid به دنبال ورودی pid میگردیم و آن را return میکنیم. همچنین تفاوت دیگر این است که مقدار waiting_processor در استراکچر proc هم برابر curproc میکنیم:



در تابع exit یک تغییر کوچک ایجاد میکنیم که متود waiting_processor که در استراکچر proc تعریف کردیم، هم از خواب بلند میکنیم:



```c
struct stat;
struct rtcdate;

// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);

int wait_for_process(int);

int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
```

در makefile، wait_for_process.c را تعریف میکنیم:

```
# Prevent deletion of intermediate files, e.g. cat.o, after first build, so
# that disk image changes after first build are persistent until clean.  More
# details:
# http://www.gnu.org/software/make/manual/html_node/Chained-Rules.html
.PRECIOUS: %.o

UPROGS=\
    _sort_string\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _wait_for_process\
```

syscall.h M | syscall.c M | sysproc.c M | defs.h M | proc.c M | wait_for_process.c U

Makefile

```
246  # CUT HERE
247  # prepare dist for students
248  # after running make dist, probably want to
249  # rename it to rev0 or rev1 or so on and then
250  # check in that version.
251
252  EXTRA=\
253      mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
254      ln.c ls.c sort_string.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
255      printf.c umalloc.c\
256      wait_for_process.c\
257      README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
258      .gdbinit.tmpl gdbutil\
259
```

تست را به صورت زیر طبق صورت پروژه تعریف میکنیم:

syscall.h M | syscall.c M | sysproc.c M | defs.h M | proc.c M | wait_for_process.c U

wait_for_process.c > wait_for_process_test(void)

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "syscall.h"

void wait_for_process_test(void) {
    int pid1, pid2;
    if((pid1 = fork())!=0) // Create a child
    {
        // We are in the parent
        if((pid2 = fork())!=0)
        {
            // We are in the parent
            wait();
            printf(1, "Second child has finished \n");
            printf(1, "Parent has been finished\n");
            exit();
        }
        else
        {
            // We are in the 2nd child
            int termpid = wait_for_process(pid1);
            if(termpid > 0){
                printf(1, "First child has finished \n");
            }
            exit();

        }
    }
    else
    {
```

```c
    else
    {
        // We are in the 1st child
        for(int i=0;i<100000000;i++){
            if(i== 9000){
                printf(1, "Wait %d iterations for first child\n", i);
            }
            else if(i== 900000){
                printf(1, "Wait %d iterations for first child\n", i);
            }
            else if(i== 9000000){
                printf(1, "Wait %d iterations for first child\n", i);
            }
            else if(i== 90000000){
                printf(1, "Wait %d iterations for first child\n", i);
            }
        }
        exit();
    }
}

int
main(int argc, char *argv[])
{
    printf(1, "Testing wait_for_process function\n");
    wait_for_process_test();

    exit();
}
```

خروجی:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

ld -m    elf_i386 -N -e main -Ttext 0 -o _wait_for_process wait_for_process.o ulib.o usys.o printf.o umalloc.o
objdump -S _wait_for_process > wait_for_process.asm
objdump -t _wait_for_process | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > wait_for_process.sym
./mkfs fs.img README _sort_string _cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _usertest
nmeta 59 (boot, super, log blocks 30 inode blocks 26, bitmap blocks 1) blocks 941 total 1000
balloc: first 732 blocks have been allocated
balloc: write bitmap block at sector 58
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,med
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
Group #22:
1. Danial Saeedi
2. Soroosh Sadeghian
3. Mohammad GharehHasanloo
$ wait_for_process
Testing wait_for_process function
Wait 9000 iterations for first child
Wait 900000 iterations for first child
Wait 9000000 iterations for first child
Wait 90000000 iterations for first child
First child has finished
Second child has finished
Parent has been finished
mylinux@ubuntu:~/Desktop/OS-Lab-main/Project1$
```

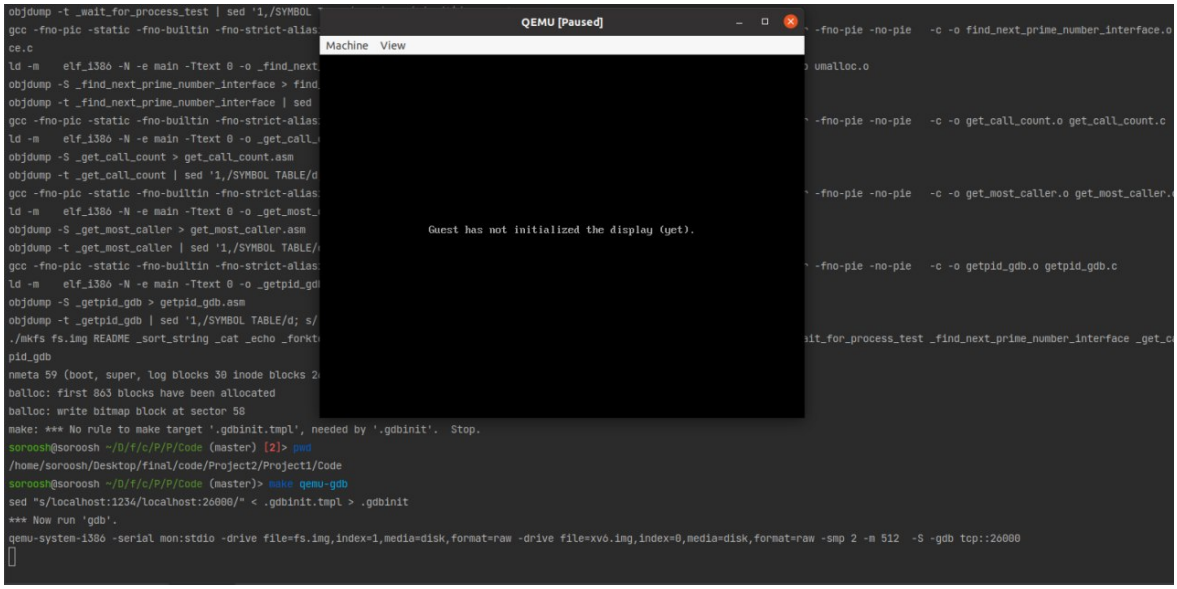
خروجی برای همه فراخوانی های سیستمی جدید

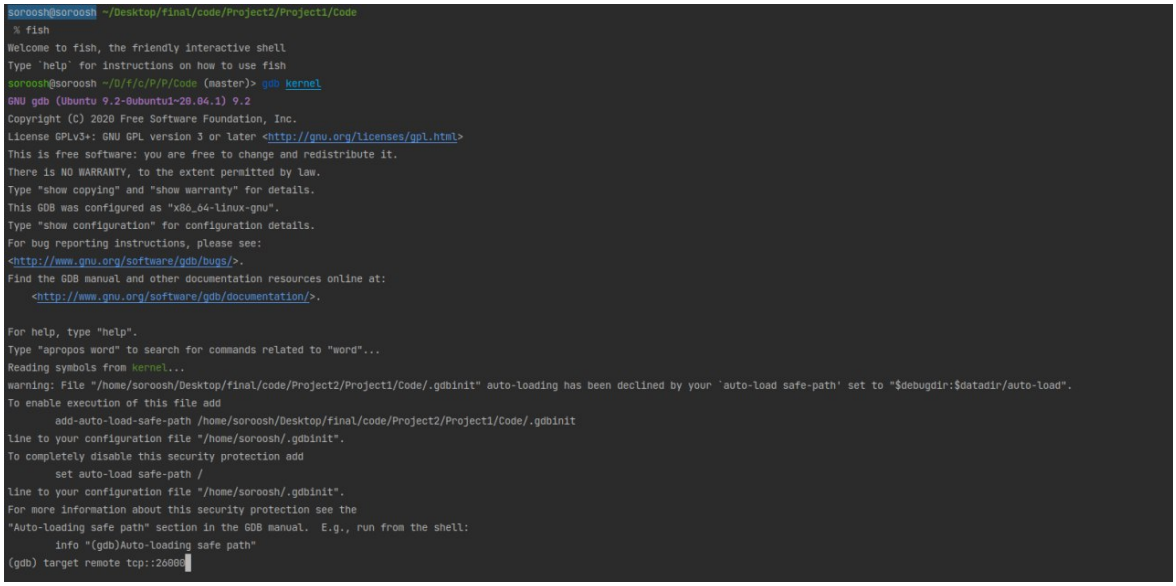
```
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap
init: starting sh
Group #22:
1. Danial Saeedi
2. Soroosh Sadeghian
3. Mohammad GharehHasanloo
$ get most caller
pid of the most caller(wait): 1
pid of the most caller(fork): 1
pid of the most caller(wrtie): 1
$ get call count
child fork count 0
child write count 19
child get call count count 3
wait count 0
parent fork count 1
parent write count 20
parent get call count count 3
wait count 1
$ get most call
exec: fail
exec get most call failed
$ get most caller
pid of the most caller(wait): 2
pid of the most caller(fork): 2
pid of the most caller(wrtie): 3
$ wait for process test
Testing wait for process function
Wait 9000 iterations for first child
Wait 900000 iterations for first child
Wait 9000000 iterations for first child
Wait 90000000 iterations for first child
First child has finished
Second child has finished
Parent has been finished
$ find next prime number 2134
Kernel: sys find next prime number() called for number 2134
The next prime number is:  2137
$ get most caller
pid of the most caller(wait): 2
pid of the most caller(fork): 2
pid of the most caller(wrtie): 9
$ get call count
child fork count 0
child write count 19
child get call count count 3
wait count 0
parent fork count 1
parent write count 20
parent get call count count 3
```

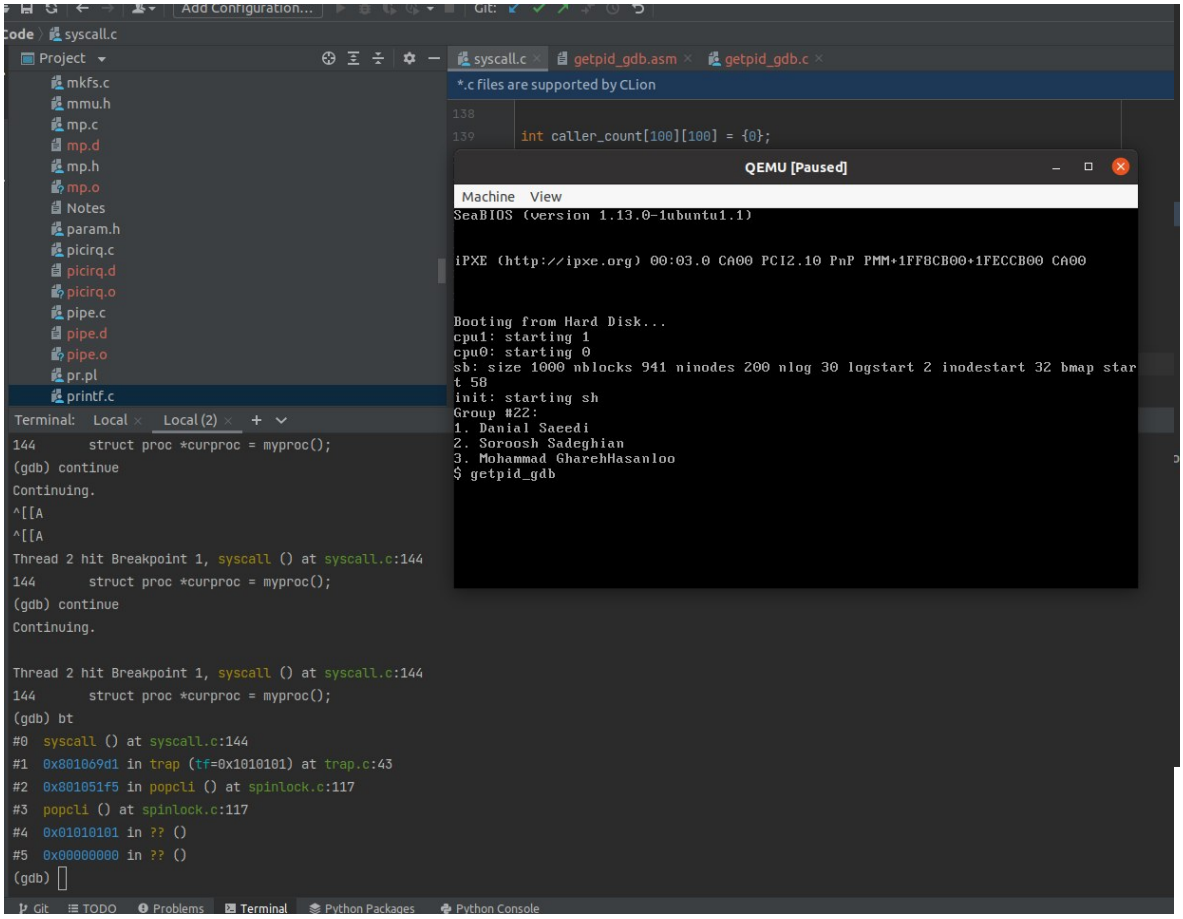
بررسی گام های اجرای فراخوانی سیستمی در سطح کرنل توسطgdb

getpid_gdb.c | Makefile | find_next_prime_number_interface.c

```c
#include "types.h"
#include "stat.h"
#include "user.h"

int main(int argc, char *argv[])
{
        printf(1, "GETPID() IS CALLED! %d\n", getpid());
        exit();
}
```




soroosh@soroosh ~/Desktop/final/code/Project2/Project1/Code
% fish
Welcome to fish, the friendly interactive shell
Type `help` for instructions on how to use fish
soroosh@soroosh ~/D/f/c/P/P/Code (master)> gdb kernel
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
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:
<http://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 kernel...
warning: File "/home/soroosh/Desktop/final/code/Project2/Project1/Code/.gdbinit" auto-loading has been declined by your `auto-load safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
add-auto-load-safe-path /home/soroosh/Desktop/final/code/Project2/Project1/Code/.gdbinit
line to your configuration file "/home/soroosh/.gdbinit".
To completely disable this security protection add
set auto-load safe-path /
line to your configuration file "/home/soroosh/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual. E.g., run from the shell:
info "(gdb)Auto-loading safe path"
(gdb) target remote tcp::26000



دستورات را مطابق عکس‌های گذاشته شده و به ترتیب برای این مرحله اجرا میکنیم و نتیجه هم در ترمینال قابل مشاهده است.