پروژه اول
درس سیستم عامل

افزودن یک SYSTEM CALL جدید به کرنل XV6

پیاده سازی برنامه کاربر RDC جهت استفاده از GETREAD SYSTEM CALL

اعضای گروه

سارا برادران

غزاله زمانی

دانشگاه صنعتی اصفهان

دانشکده برق و کامپیوتر

ترم اول سال تحصیلی ۱۴۰۰ - ۱۳۹۹

- فرآیند اجرای یک system call در داخل xv6

قطعه کد زیر که در فایل usys.S موجود است، macro نامیده می شود،در حقیقت می توان macro را همانند تابع دانست و کامپایلر پیش از عمل کامپایل اطلاعات مربوطه را داخل کد اسمبلی جایگذاری می کند.

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```

برای مثال بررسی خواهیم کرد که getpid system call چگونه در داخل marco جایگذاری می شود:

آنچه در داخل این کد اسمبلی رخ می دهد این است که ابتدا یک global name تحت عنوان getpid تعریف شده و سپس این get-pid شماره اختصاصی getpid system call که در فایل syscall.h تعریف شده است را در داخل رجیستر %eax قرار داده و سپس یک interrupt را با شماره T_SYSCALL فراخوانی می کند. در حقیقت T_SYSCALL مقدار ۶۴ خواهد بود و مانند تصویر زیر در فایل traps.h شماره های نظیر هر interrupt تعیین شده است.

```
// These are arbitrarily chosen, but with care not to overlap
// processor defined exceptions or interrupt vectors.
#define T_SYSCALL       64      // system call
#define T_DEFAULT      500      // catchall
```

آنچه ضمن فراخوانی INT $64 یا همان INT %T_SYSCALL رخ می دهد این است که پس از انجام یک سری اعمال سخت افزاری به دستور vector64 در داخل فایل vector.S پرش خواهد کرد و ابتدا اعداد ۰ و ۶۴ را داخل stack ذخیره و سپس به تابع alltraps موجود در فایل trapasm.S پرش کرده و trapframe را خواهد ساخت و trapno مربوطه همان عدد ۶۴ ذخیره شده در stack خواهد بود.

```
#!/usr/bin/perl -w

# Generate vectors.S, the trap/interrupt entry points.
# There has to be one entry point per interrupt number
# since otherwise there's no way for trap() to discover
# the interrupt number.

print "# generated by vectors.pl - do not edit\n";
print "# handlers\n";
print ".globl alltraps\n";
for(my $i = 0; $i < 256; $i++){
    print ".globl vector$i\n";
    print "vector$i:\n";
    if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
        print "  pushl \$0\n";
    }
    print "  pushl \$$i\n";
    print "  jmp alltraps\n";
}

print "\n# vector table\n";
print ".data\n";
print ".globl vectors\n";
print "vectors:\n";
for(my $i = 0; $i < 256; $i++){
    print "  .long vector$i\n";
}
```

تصویر فوق محتویات فایل vector.pl را نمایش می دهد که سازنده فایل vector.S به صورت زیر خواهد بود:

```
# sample output:
#   # handlers
#   .globl alltraps
#   .globl vector0
#   vector0:
#     pushl $0
#     pushl $0
#     jmp alltraps
#   ...
#
#   # vector table
#   .data
#   .globl vectors
#   vectors:
#     .long vector0
#     .long vector1
#     .long vector2
#   ...
```

تصویر زیر محتویات فایل trapasm.S را نمایش می دهد

```
#include "mmu.h"

  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

پس از کامل کردن trapframe و ذخیره سازی رجیستر هایی همچون (eax,ecs,...,esi,edi) تابع trap() موجود در فایل trap.c فراخوانی شده و اشاره گری به ابتدای trapframe ساخته شده را به این تابع به عنوان آرگومان ورودی پاس می دهیم و در انتها پس از اتمام تابع trap() مجددا رجیستر های مذکور در تابع trapret موجود در همان فایل trapasm.S بازنشانی خواهند شد.

```
//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
```

ضمن ورود به تابع trap پس از ساختن trapframe ابتدا بررسی می شود اگر trapno همان ۶۴ یعنی شماره اختصاصی interrupt مربوط به system call ها بود آنگاه myproc()->tf برابر همان اشاره گر به ابتدای trapframe قرار خواهد گرفت و سپس تابع syscall موجود در فایل syscall.c فراخوانی خواهد شد. همانطور که در تصویر زیر مشخص است این تابع ابتدا مقدار رجیستر eax در trapframe را مورد بررسی قرار داده و اگر این مقدار جزو مقادیر اختصاص یافته به system call ها (۱ تا ۲۱) بود آنگاه تابع نظیر system call مربوطه که در فایل sysproc.c است را فراخوانی خواهد کرد. نحوه فراخوانی این تابع به وسیله آرایه ای از function pointer هاست که در داخل همان فایل syscall.c موجود بوده و هر عضو از این آرایه یک اندیس را داراست که این اندیس همان شماره اختصاصی هر system call است که در داخل فایل syscall.h تعریف شده. نهایتا مقدار بازگشتی حاصل از فراخوانی تابع نظیر system call مجددا در رجیستر eax ذخیره خواهد شد.

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  if(num == 5)
    readcount++;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

```
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[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_getreadcount]   sys_getreadcount,
};
```

- مراحل لازم جهت افزودن sytem call :

جهت اضافه کردن system call مذکور نیاز است در فایل های زیر تغییراتی ایجاد نماییم:

Syscall.h
Syscall.c
Sysproc.c
Usys.S
User.h
Defs.h

به عنوان اولین گام یک متغیر تحت عنوان readcount را به نحوی که در تصویر زیر نشان داده شده در فایل defs.h اضافه می نماییم:
(از این متغیر در فایل های دیگر و جهت شمارش تعداد فراخوانی های read system call استفاده خواهیم کرد.)

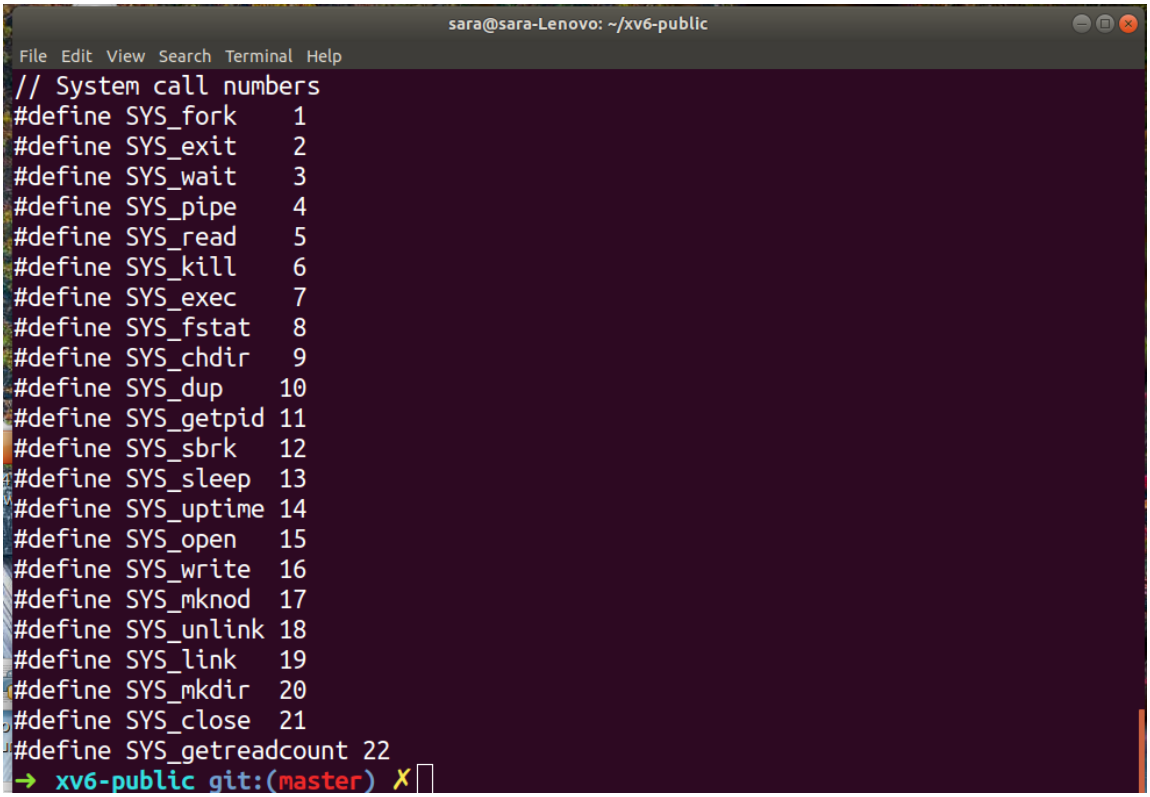
```
→ xv6-public git:(master) ✗ cat defs.h
struct buf;
struct context;
struct file;
struct inode;
struct pipe;
struct proc;
struct rtcdate;
struct spinlock;
struct sleeplock;
struct stat;
struct superblock;

extern int readcount;

// bio.c
void            binit(void);
struct buf*     bread(uint, uint);
void            brelse(struct buf*);
void            bwrite(struct buf*);
```

سپس لازم است system call مربوطه را به همراه شماره ای که قصد داریم به آن اختصاص دهیم در فایل syscall.h اضافه کنیم. همانطور که از محتویات این فایل در تصویر زیر مشخص است تمام system call های xv6 که تعداد آن ها در مجموع ۲۱ عدد است به همراه شماره اختصاصی مربوط به هر system call در داخل این فایل قرار دارد. برای مثال system call شماره ۵ همان read و system call شماره ۱۱ همان getpid می باشد. در این قسمت شماره ۲۲ را به system call جدید یعنی getreadcount با افزودن خط زیر اختصاص می دهیم:

```
#define SYS getreadcount 22
```



در گام بعد لازم است یک اشاره گر به system call جدید را در فایل syscall.c اضافه نماییم. این فایل حاوی آرایه ای از function pointer ها می باشد به این صورت که هر عضو از آرایه ، اشاره گری به system call با شماره اختصاصی قرار گرفته در [ ] خواهد بود.کافی است در انتهای این آرایه دستور زیر را اضافه نماییم:

```
[SYS_getreadcount] sys_getreadcount
```

- در حقیقت زمانی که system call با شماره اختصاصی k رخ می دهد تابعی که function pointer مربوطه، به آن اشاره می کند فراخوانی شده و لذا اعمالی که در این system call باید انجام شود اعمال می گردند.

```
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[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_getreadcount]   sys_getreadcount,
};
```

سپس در داخل همان فایل syscall.c نیاز است prototype مربوط به system call جدید را نیز اضافه نماییم. با افزودن خط زیر می توان این کار را انجام داد:

```
extern int sys_getreadcount(void)
```

- همانطور که در تصویر زیر مشخص است این فایل حاوی prototype مربوط به تمام ۲۱ مورد system call موجود در xv6 می باشد. به علاوه همانطور که در prototype مشخص کردیم این system call هیچ آرگومان ورودی ای دریافت نمی کند و تنها یک عدد integer که همان تعداد فراخوانی های read system call است را به عنوان خروجی باز می گرداند.

```
extern int sys_chdir(void);
extern int sys_close(void);
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
extern int sys_mkdir(void);
extern int sys_mknod(void);
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_getreadcount(void);
```

به علاوه در داخل فایل syscall.c تابعی تحت عنوان void syscall(void) موجود می باشد. در حقیقت این تابع هنگام فراخوانی یک system call در سیستم فراخوانی خواهد شد،سپس بنابر اینکه system call فراخوانی شده از چه نوعی است (با توجه به مقدار رجیستر eax که همان شماره system call مربوطه را در بر دارد) به وسیله function pointer مربوط به آن system call ، تابع مربوطه را فراخوانی کرده و مقدار بازگشتی آن را در رجیستر eax ذخیره می نماید. از آنجایی که قصد شمارش تعداد فراخوانی های read system call را داریم، کافی است در این تابع اگر شماره system call فراخوانی شده برابر ۵ بود متغیر readcount را یک واحد اضافه کنیم. جزییات مربوط به این تغییر را در تصویر زیر می توان مشاهده کرد:

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  if(num == 5)
    readcount++;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
→ xv6-public git:(master) ✗
```

همچنین لازم است در ابتدای فایل syscall.c مقدار اولیه متغیر readcount را مانند آنچه در تصویر زیر مشخص است برابر صفر قرار دهیم زیرا در غیر این صورت ممکن است یک مقدار اولیه ناصفر در متغیر readcount موجود باشد که شمارش را با خطا مواجه کند:

```
→ xv6-public git:(master) ✗ cat syscall.c
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
#include "syscall.h"

int readcount = 0;

// User code makes a system call with INT T_SYSCALL.
// System call number in %eax.
// Arguments on the stack, from the user call to the C
// library system call function. The saved user %esp points
// to a saved program counter, and then the first argument.
```

در گام بعد می بایست بدنه تابع مربوط به آن system call که prototype آن را در قسمت قبل اضافه کردیم ، پیاده سازی کنیم. به این منظور در فایل sysproc.c خطوط برنامه زیر را تحت عنوان یک تابع اضافه خواهیم کرد:

```
int
sys_getreadcount(void)
{
        return readcount;
}
```

```
// return how many clock tick interrupts have occurred
// since start.
int
sys_uptime(void)
{
  uint xticks;

  acquire(&tickslock);
  xticks = ticks;
  release(&tickslock);
  return xticks;
}
int
sys_getreadcount(void)
{
  return readcount;

}
```

در مرحله آخر برای افزودن system call ، لازم است دو فایل usys.S , user.h را تغییر دهیم. این دو فایل شامل interface هایی است که امکان دسترسی به system call های جدید را به برنامه های کاربر می دهد.
ابتدا در فایل usys.S خط زیر را اضافه می کنیم:

```
SYSCALL(getreadcount)
```

```
SYSCALL(fork)
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(getreadcount)
→ xv6-public git:(master) ✗
```

سپس لازم است prototype مربوط به تابعی که امکان استفاده از system call جدید را در اختیار برنامه کاربر قرار می دهد در داخل فایل user.h اضافه نماییم. این کار را با افزودن خط زیر انجام می دهیم و prototype تعریف شده بدان معناست که user program ها برای استفاده از getreadcount system call می توانند تابع ;(getreadcount را فراخوانی نمایند که در نهایت این تابع یک عدد integer را باز می گرداند که همان تعداد فراخوانی های read system call است:

```
int getreadcount(void);
```

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

// ulib.c
int stat(const char*, struct stat*);
```

- مراحل لازم جهت افزودن برنامه rdc :

حال قصد داریم یک user program تحت عنوان rdc برای استفاده از getreadcount system call بنویسیم، این برنامه را با برداشت ایده از برنامه $cat.c$ به شکل زیر و با استفاده از زبان C پیاده سازی نموده ایم:

```
→ xv6-public git:(master) ✗ cat rdc.c
#include "types.h"
#include "stat.h"
#include "user.h"

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

  if(argc > 1){
    printf(1, "rdc: too many arguments\n");
    exit();
  }

  printf(1, "Hi, the number of read syscall is %d so for!\n", getreadcount());
  exit();
}
→ xv6-public git:(master) ✗
```

از آنجایی که جهت اجرای برنامه rdc تنها آرگومان لازم نام برنامه است لذا اگر argc > 1 آنگاه user program خطای too many arguments را چاپ کرده و خاتمه می یابد.
سپس لازم است در Makefile تغییرات زیر را ایجاد نماییم.
مطابق شکل زیر در بخش UPROGS و EXTRA نام user program نوشته شده یعنی rdc را اضافه می کنیم:

```
EXTRA=\
        mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
        ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
        rdc.c\
        printf.c umalloc.c\
        README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
        .gdbinit.tmpl gdbutil\
```

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

سپس یک بار دستور make clean را در ترمینال اجرا می کنیم تا فایل های اجرایی ایجاد شده حذف گردند و مجددا با استفاده از دستور make qemu-nox کرنل xv6 ساخته شده و در محیط qemu اجرا می شود. حال اگر دستور ls را وارد کنیم برنامه rdc به برنامه های موجود اضافه شده است که می توان با وارد کردن دستور rdc یا rdc./ آن را اجرا و نتیجه را مشاهده کرد.

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 13644
echo           2 4 12652
forktest       2 5 8088
grep           2 6 15520
init           2 7 13240
kill           2 8 12704
ln             2 9 12604
ls             2 10 14788
mkdir          2 11 12784
rm             2 12 12764
sh             2 13 23248
stressfs       2 14 13432
usertests      2 15 56364
wc             2 16 14184
zombie         2 17 12428
rdc            2 18 12576
console        3 19 0
$ rdc
Hi, the number of read syscall is 40 so for!
$
```

- نتیجه تست :

نتیجه تست getreadcount system call نیز در تصویر زیر قابل مشاهده است:

```
File  Edit  View  Search  Terminal  Help
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32
-Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -o wc
.o wc.c
ld -m    elf_i386 -N -e main -Ttext 0 -o _wc wc.o ulib.o usys.o printf.o umalloc
.o
objdump -S _wc > wc.asm
objdump -t _wc | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > wc.sym
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32
-Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -o zo
mbie.o zombie.c
ld -m    elf_i386 -N -e main -Ttext 0 -o _zombie zombie.o ulib.o usys.o printf.o
 umalloc.o
objdump -S _zombie > zombie.asm
objdump -t _zombie | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > zombie.sym
./mkfs fs.img README _cat _echo _forktest _grep _init _kill _ln _ls _test_1 _tes
t_2 _mkdir _rm _sh _stressfs _usertests _wc _zombie
nmeta 59 (boot, super, log blocks 30 inode blocks 26, bitmap blocks 1) blocks 94
1 total 1000
balloc: first 626 blocks have been allocated
balloc: write bitmap block at sector 58

test 1: passed
test 2: passed
→ initial-xv6 git:(master) ✗
```

- روش دوم :

به نحو دیگری نیز می توان عمل شمارش read system call ها را انجام داد. این روش مستلزم ایجاد تغییرات اندکی در read sys-tem call می باشد. به این صورت که در درون تابع SYS_read موجود در فایل sysfile.c متغیر readcount را یک واحد اضافه کرده و در ابتدای همین فایل مقدار اولیه متغیر را برابر صفر قرار می دهیم. به این نحو دیگر نیازی به ایجاد تغییرات در تابع syscall موجود در فایل syscall.c و مقدار دهی اولیه متغیر در این فایل نمی باشد.(در فایل های xv6 ضمیمه شده از روش اول استفاده کرده و خط های مربوط به روش دوم در فایل sysfile.c کامنت شده است.)

```
→ xv6-public git:(master) ✗ cat sysfile.c
//
// File-system system calls.
// Mostly argument checking, since we don't trust
// user code, and calls into file.c and fs.c.
//

#include "types.h"
#include "defs.h"
#include "param.h"
#include "stat.h"
#include "mmu.h"
#include "proc.h"
#include "fs.h"
#include "spinlock.h"
#include "sleeplock.h"
#include "file.h"
#include "fcntl.h"
int readcount = 0;
// Fetch the nth word-sized system call argument as a file descriptor
// and return both the descriptor and the corresponding struct file.
```

```
int
sys_read(void)
{
  struct file *f;
  int n;
  char *p;
  readcount++;
  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return fileread(f, p, n);
}
```