گزارش ازمایشگاه سیستم عامل پروژه 2

سبحان کوشکی محمد جزایری نوید هاشمی
810101496 810101399 810101549

سوال 1: کتابخانه‌های سطح کاربر در xv6، برای ایجاد ارتباط میان برنامه های کاربر و کرنل به کار میروند. این کتابخانه ها شامل توابعی هستند که از فراخوانی های سیستمی استفاده میکنند تا دسترسی به منابع سخت افزاری و نرم افزاری سیستم عامل ممکن شود. با تحلیل فایل های موجود در متغیر ULIB در xv6، توضیح دهید که چگونه این کتابخانه ها از فراخوانی های سیستمی بهره میبرند؟ همچنین، دلایل استفاده از این فراخوانی ها و تأثیر آنها بر عملکرد و قابلیت حمل برنامه ها را شرح دهید.

در متغیر ULIB در makefile چهار تا .o هستند که در شکل زیر ان ها را میبینید

```
146    ULIB = ulib.o usys.o printf.o umalloc.o
```

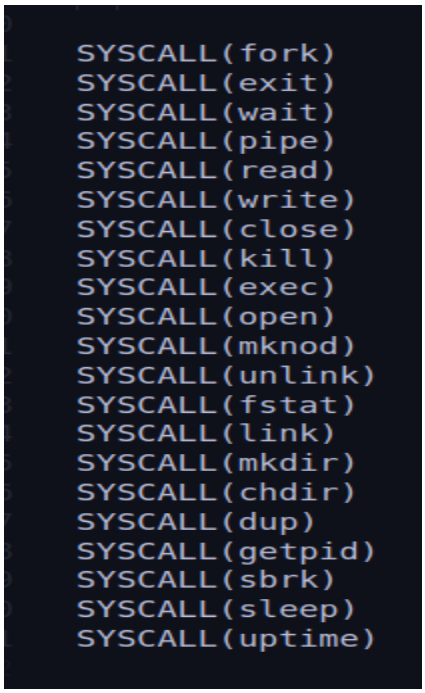
که در زیر هر یک را توضیح میدهیم که چه کاری انجام میدهند و چگونه از فراخوانی های سیستمی استفاده میکنند

در ulib.o که از کد ulib.c امده است ما توابعی میبینیم که که در بعضی از ان ها از systemcall ها بهره برده شده است مانند read در gets یا open close در stat میدانیم که این چند systemcall ها در کرنل قرار دارند و ما با استفاده از این دستورات میتوانیم از خدمات کرنل استفاده کنیم

در usys.h یا user.h میتوانیم این systemcall هایی که گفته ایم را لیست ان ها را ببینید. که در usys.h ما چندین wrapper برای systemcall ها به زبان اسمبلی داریم در printf.o ما باید به کد ان یعنی printf.c برویم در اینجا هر چند تابع برای نوشتن داریم که همه ان ها به putc برمیگرددند که در putc ما systemcall نوشتن write را میبینیم که باز این دستور یک دستور از لایه کرنل است

در umalloc.o به سراغ کد umalloc.c میرویم در این مد هم در بخش morecore از یک systemcall به نام sbrk استفاده کردیم که این هم جزو systemcall هایی است که برای افزایش فضای process استفاده میشود

در دستورات زیر که در usys.S است لیست تابع های system call را میبینید



همانطور که در صورت پروژه به ان اشاره شده است ما از این توابع استفاده میکنیم به دلیل اینکه بعضی از کاربرد هایی که میخواهیم از سخت افزار بهره بگیریم تابع هایی هستند مه فقط os به ان ها دسترسی دارند و ما باید با این دستورات به os دسترسی داشته باشی که او خدمات را به ما ارایه کند همچنین به دلیل استفاده از این system call ها ما میتوانیم برناه های خودمان را بر روی تعداد زیادی دیگر از پردازنده ها و ... برنامه خود را اجرا کنیم در هر computer این دستورات کد های متفاوتی دارند و ما اگر میخواستیم دستی این مار ها را انجام دهیم غیرممکن بود با اینکار ما این وظیفه را بر روی خود os ان سخت افزار قرار میدهیم

پرسش 2:فراخوانی های سیستمی تنها روش برای تعامل برنامه های کاربر با کرنل نیستند. چه روش های دیگری در لینوکس وجود دارند که برنامه های سطح کاربر می توانند از طریق آنها به کرنل دسترسی داشته باشند؟ هر یک از این روش ها را به اختصار توضیح دهید.

- سوکت socket: سوکت‌های Netlink یک نوع سوکت خاص هستند که برای ارتباط بین کرنل و برنامه‌های سطح کاربر طراحی شده‌اند. از این سوکت‌ها برای پیکربندی تنظیمات شبکه، ارسال و دریافت رویدادهای شبکه، و ارتباطات درون‌سیستمی استفاده می‌شود.
- سیگنال signal: سیگنال‌ها پیام‌های ناهمزمانی هستند که از طرف کرنل (یا فرآیند دیگری) به فرآیند کاربر ارسال می‌شوند. از سیگنال‌ها می‌توان برای اطلاع‌رسانی به برنامه‌های کاربر درباره رویدادهایی مانند پایان زمان تایمر، دسترسی به ورودی/خروجی، یا خطاها استفاده کرد.
- Memory-Mapped Files : با استفاده از تابع `mmap`، یک فایل یا دستگاه به حافظه برنامه نگاشت می‌شود و سپس برنامه می‌تواند مستقیماً به داده‌های فایل دسترسی داشته باشد، بدون اینکه نیاز به فراخوانی‌های سیستمی متعدد برای خواندن یا نوشتن داده باشد.

سوال 3:آیا باقی تله ها را نمی توان با سطح دسترسی DPL USER فعال نمود؟ چرا؟
خیر نباید این کار صورت بگیرد و نباید به ان اجاره داده شود دسترسی DPL USER برای کاربر است اگر کسی با DPL USER به بقیه تله ها دسترسی داشته باشد کاربر به تمام امکانات و خدمات OS میتواند دسترسی داشته باشد و این یعنی که ما protection را دیگر در ان سیستم نداریم و دیگر بخش user با kernel تفاوتی نداشت و این را ما نمیخواهیم
اینکار باعث افت شدید امنیت در سیستم میشود و کابر یا یک عامل خارجی میتواند ان سیستم را مورد هدف قرار دهد

سوال 4:در صورت تغییر سطح دسترسی ، ss و esp روی پشته Push میشود. در غیر اینصورت push نمیشود. چرا؟
وقتی ss و esp را روی استک push میکنیم که میخواهیم سطح کاربر را از user mode به kernel mode ببریم و اینکار را انجام میدهیم که بدانیم که سطح دسترسی را عوض کرده ایم و بعد از pop کردن ان ها میتوانیم ادامه روند برنامه را پیش بگیریم ولی اگر نخواهیم که این تغییر را در سطح دسترسی انجام دهیم نیاز به push کردن این دو در استک نداریم

سوال 5:در مورد توابع دسترسی به پارامترهای فراخوانی سیستمی به طور مختصر توضیح دهید. چرا در argptr() بازه آدرس ها بررسی میگردد؟ تجاوز از بازه معتبر، چه مشکل امنیتی ایجاد میکند؟ در صورت عدم بررسی باز ها در این تابع، مثالی بزنید که در آن، فراخوانی سیستمی (sys_read) اجرای سیستم را با مشکل روبرو سازد.
سه تابع داریم که برای access به پارامتر های system call ها نوشته شده اند

```
// syscall.c
int             argint(int, int*);
int             argptr(int, char**, int);
int             argstr(int, char**);
```

argint: که دو ورودی میگیرد وردی اول شماره پارامتری است که میخواهد ان را بخواند و دومی را یک حافظه int به ان میدهد که مقدار اون پارامتر را در ان بریزد این کار اگر موفقیت امیز بود عدد 0 را برمیگرداند و اگر این کار با موفقیت انجام نشود عدد منفی 1 را برمیگرداند

```
// Fetch the nth 32-bit system call argument.
int
argint(int n, int *ip)
{
  return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
}
```

argptr: که 3 ورودی دارد ورودی اول شماره پارامتر که میخواهد را مینویسیم که ای کار با استفاده از تابع argint که در بالا توضیح دادیم اتفاق می افتد ورودی دوم به صورت رفرنس به این تابع میدهیم که محتواهایی که میخواهد را در این حافظه بریزد و ورودی سوم اندازه سایز را میدهیم مانند تابع argint هم اگر موفقیت امیز باشد 0 و در غیر اینصورت منفی 1 برمیگرداند

```
// Fetch the nth word-sized system call argument as a pointer
// to a block of memory of size bytes.  Check that the pointer
// lies within the process address space.
int
argptr(int n, char **pp, int size)
{
  int i;
  struct proc *curproc = myproc();

  if(argint(n, &i) < 0)
    return -1;
  if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)
    return -1;
  *pp = (char*)i;
  return 0;
}
```

argstr: در این تابع ما شماره ارگومانی که میخواهیم از ان بخوانیم را به ان میدهیم و او این چیزی که میخواهیم را در char* قرار میدهد

```
// Fetch the nth word-sized system call argument as a string pointer.
// Check that the pointer is valid and the string is nul-terminated.
// (There is no shared writable memory, so the string can't change
// between this check and being used by the kernel.)
int
argstr(int n, char **pp)
{
  int addr;
  if(argint(n, &addr) < 0)
    return -1;
  return fetchstr(addr, pp);
}
```

argfd: سه ورودی میگیرد که ان ها به ترتیب شماره پارامتر ، pointer به file pointer ، descriptor به ادرس ساختمان داده فایل است که در پایان کار این تابع چیزی که میخواهیم مقدار دهی میکنیم در صورت موفقیت 0 غیر این صورت منفی 1 برمیگرداند

```
// Fetch the nth word-sized system call argument as a file descriptor
// and return both the descriptor and the corresponding struct file.
static int
argfd(int n, int *pfd, struct file **pf)
{
  int fd;
  struct file *f;

  if(argint(n, &fd) < 0)
    return -1;
  if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)
    return -1;
  if(pfd)
    *pfd = fd;
  if(pf)
    *pf = f;
  return 0;
}
```

اگر در systemcall sys_read بازه را چک نکنیم ممکن است از یک فایل دیگری بخوانیم یا اصلا یک مقدار نا معتبر بخوانیم و اینکار ممکن است باعث مشکلاتی برای دیگر پردازه ای که در کنار این پردازه اجرا میشود شود

```c
int
sys_read(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return fileread(f, p, n);
}
```

بررسی گام های اجرای فراخوانی سیستمی در سطح کرنل توسط GDB

در ابتدا برنامه ای که خواسته شده را مینویسیم که در ان از تابع getpid استفاده کرده ایم

```c
//this for type int char
#include"types.h"

//this for we can use getpid()
#include"user.h"

int main(int argc , char*argv[]) {
    int pid = getpid();
    printf(1 , "hi :)\nwe can getpid and this is the pid : %d\n" , pid);
    exit();
}
```

در این برنامه ساده pid را میگیریم و در متنی مینویسیم
حالا باید این برنامه را مانند ازمایش قبلی در makefile اضافه کنیم
پس از این کار ها نوبت به gdb میرسد دستور make qemu-gdb را وارد کرده سپس در ترمینال دیگر می ایم و دستور gdb kernel را اجرا میکنیم اگر در مود kernel نباشیم نمیتوانیم به کد های syscall دسترسی پیدا کنیم
پس از این موارد مانند ازمایش قبل باید gdb را به qemu وصل کنیم با دستور target remote tcp::26000 این کاررا انجام میدهیم.
سپس برای گذاشتن breakpoint در اول تابع syscall دستور b syscall.c:136 را میزنیم
حالا دستورات بعدی را وارد میکنیم

```
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 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:
<https://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/sobhankj/sobhan/.gdbinit" auto-loading has been declined by your `auto-load safe-path' set to "$debu
gdir:$datadir/auto-load".
To enable execution of this file add
        add-auto-load-safe-path /home/sobhankj/sobhan/.gdbinit
line to your configuration file "/home/sobhankj/.config/gdb/gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/home/sobhankj/.config/gdb/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
Remote debugging using tcp::26000
0x0000fff0 in ?? ()
(gdb) b syscall.c:136
Breakpoint 1 at 0x80105e6e: file syscall.c, line 137.
(gdb) r
The "remote" target does not support "run".  Try "help target" or "continue".
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137       num = curproc->tf->eax;
```

وقتی دستور bt را میزنیم به ما چنین خط هایی خروجی میدهد

```
(gdb) bt
#0  syscall () at syscall.c:137
#1  0x80106ead in trap (tf=0x8dffffb4) at trap.c:43
#2  0x80106c4f in alltraps () at trapasm.S:20
```

در واقع دستور bt برای نشان دادن stack call است

خود stack call برای ذخیره کردن مراحل اجرای برنامه است و میتوانیم با داشتن این stack بفهمیم که برنامه ما درست اجرا میشود یا خیر

ما با زدن bt میتوانیم محتویات stack call را ببینیم به عنوان مثال یک trap و یک alltrap در فایل هایی که نمایش داده شده است صورت گرفته است

برای دستورات بعدی باید بفهمیم که در stack call چه چیز هایی وجود دارد و چگونه به ان اضافه میشوند

وقتی ما برنامه ای را اجرا میکنیم در درس متوجه شدیم که stack برای ان درست میشود که اطلاعات ان از قبلی متغیر ها جایی که الان برنامه در ان است و ... در ان وجود دارد

وقتی که این تابع صدا زده میشود ما این stack را در بالای stack call میگذاریم

حالا با دستورات down و up میتوانیم در بین این stack هایی که در stack call وجود دارند بالا و پایین برویم و محتویات ان ها را ببینیم

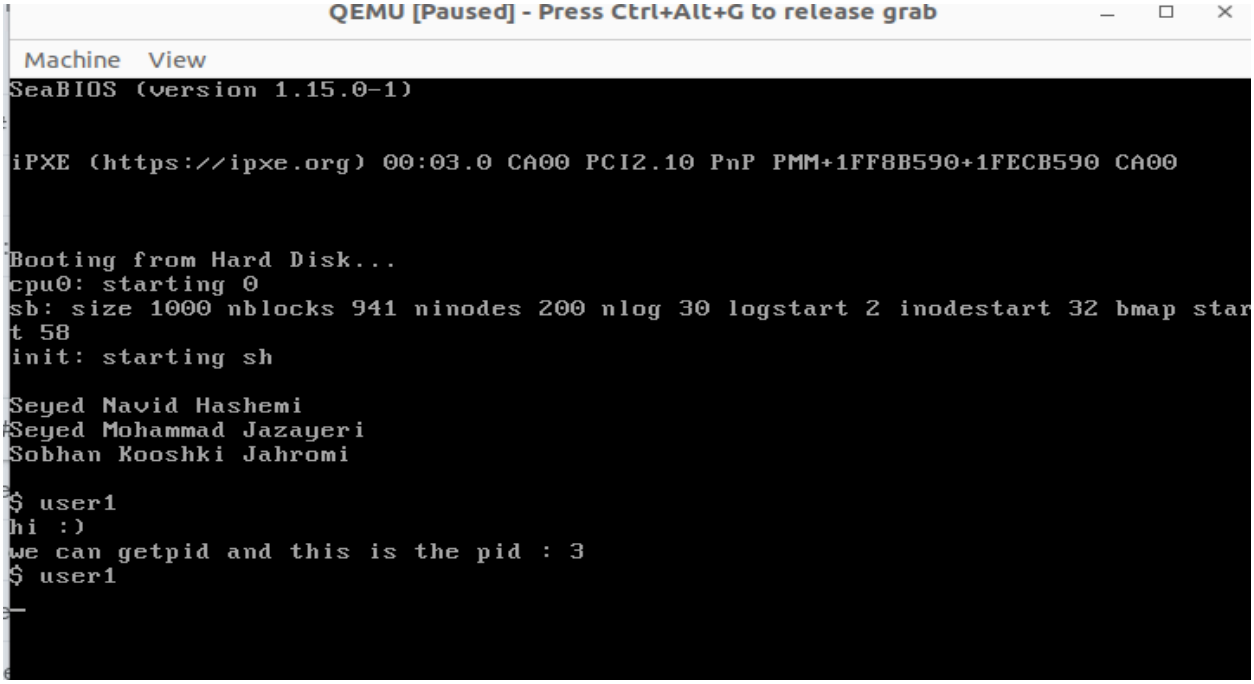
```
(gdb) down
Bottom (innermost) frame selected; you cannot go down.
(gdb) up
#1  0x80106ead in trap (tf=0x8dffffb4) at trap.c:43
43          syscall();
```

چون تابعی که در ان هستیم syscall هنوز خود چیزی را صدا نزده است پس وقتی down را در command line میزنیم چیزی برای نمایش ندارد ولی وقتی up میزنیم به دستوری میرویم که خود syscall را صدا زده است و میتوانیم ان را ببینیم

رجیستر eax برای ذخیره شماره سیستم کال است و وقتی به سراغ شماره سیستم کال ها میرویم میبینیم که getpid شماره 11 را دارد ولی ما با چاپ کردن eax در همان اول به این شماره نمیرسیم چون قبل از ان باید چندین مرحله دیگر پیش برویم تا به getpid برسیم مثل exec , sbrk , wait , fork , read و ممکن است چندین دستور دیگر هم باشد و سپس به دستور getpid برسیم

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137         num = curproc->tf->eax;
(gdb) print myproc()->tf->eax
$11 = 5
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137         num = curproc->tf->eax;
(gdb) print myproc()->tf->eax
$12 = 1
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137         num = curproc->tf->eax;
(gdb) print myproc()->tf->eax
$13 = 3
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137         num = curproc->tf->eax;
(gdb) print myproc()->tf->eax
$14 = 12
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137         num = curproc->tf->eax;
(gdb) print myproc()->tf->eax
$15 = 7
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:137
137         num = curproc->tf->eax;
(gdb) print myproc()->tf->eax
$16 = 11
(gdb)
```

پس از تلاش های بسیار به عدد 11 رسیدیم :)



برای اینکه به دستور getpid برسیم باید برنامه ای که نوشته ایم را در زمانی که اسم های خودمان را نوشت و به روی eax = 5 رفت در ترمینال بنویسیم و سپس باز c بزنیم تا به getpid برسیم.

ارسال آرگومان های فراخوانی های سیستمی

در ادامه پروژه ما باید سیستم کال هایی به سیستم کال هایی که وجود دارد اضافه کنیم برای اینکار باید اسم این سیستم کال ها را چندین جا اضافه کنیم تا بتوانیم ان ها را در ادامه اضافه کنیم و کد ان ها را بزنیم از جاهایی که اسم ان ها را مینویسیم syscall.h است

```
#define SYS_create_palindrome 22
#define SYS_move_file 23
#define SYS_get_most_invoked_syscall 24
#define SYS_sort_syscalls 25
#define SYS_list_all_processes 26
```

در این قسمت عدد هایی هم به ان ها نسبت میدهیم که 21 تای اول برای دستوراتی هستند که وجود دارند پس ما باید از 21 به بعد اعداد خود را بنویسیم

جاهای دیگری که باید دستورات را به کرنل بشناسیم و ان ها را معرفی کنیم در syscall.c و defs.h و usys.S و user.h است در این دو قسمت هم اضافه میکنیم در syscall.c و user.h ما prototype تابع سیستم کال را اضافه میکنیم و ان سیستم کال را به ارایه syscalls اضافه میکنیم.

```
extern int sys_create_palindrome(void);
extern int sys_move_file(void);
extern int sys_get_most_invoked_syscall(void);
extern int sys_sort_syscalls(void);
extern int sys_list_all_processes(void);
```

```
[SYS_create_palindrome]   sys_create_palindrome,
[SYS_move_file]     sys_move_file,
[SYS_get_most_invoked_syscall]    sys_get_most_invoked_syscall,
[SYS_sort_syscalls]   sys_sort_syscalls,
[SYS_list_all_processes] sys_list_all_processes,
```

```
int             create_palindrome(int);
int             get_most_invoked_syscall(int);
int             sort_syscalls(int);
void            list_all_processes(void);
```

```
int create_palindrome(int);
int move_file(const char*, const char*);
int get_most_invoked_syscall(int);
int sort_syscalls(int);
void list_all_processes(void);
```

```
SYSCALL(create_palindrome)
SYSCALL(move_file)
SYSCALL(get_most_invoked_syscall)
SYSCALL(sort_syscalls)
SYSCALL(list_all_processes)
```

حال به سراق پیاده سازی تابع palintrome میرویم.
در این بخش از ما خواسته شده است که که با استفاده از رجیستر ها ارگومان ورودی تابع را پاس دهیم برای این کار ابتدا باید چیزی که میخواهیم پاس دهیم را در یک رجیستر بنویسیم این کار را با استفاده از کد اسمبلی زیر انجام میدهیم.

```
int last_ebx_value;
int number = atoi(argv[1]);

asm volatile (
    "movl %%ebx, %0;"
    "movl %1, %%ebx;"
    :"=r" (last_ebx_value)
    :"r" (number)
);
printf(1, "create_palindrome is called for %d \n", number);
int answer = create_palindrome(number);
printf(1, "The palindrome of number %d is: %d\n", number, answer);

asm("movl %0, %%ebx"
        :
        : "r"(last_ebx_value)
);
```

حال به بخش create_palindrome در sysproc.c میرویم در اینجا ان رجیستر که در ان داده را داده ایم را خوانده و به تابع اصلی پاس میدهیم

```
int
sys_create_palindrome(void){
  int n = myproc()->tf->ebx;
  cprintf("KERNEL: sys_create_palindrome(%d)\n", n);
  return create_palindrome(n);
}
```

در تابع اصلی هم عملیات palidrome را انجام داده ایم که به صورت زیر است

```
int create_palindrome(int num){
  int reversed = 0;
  int temp = num;
  int num_of_digits = 0;

  while (temp > 0)
  {
      reversed = reversed * 10 + (temp % 10);
      temp /= 10;
      num_of_digits += 1;
  }

  int palindrome = reversed;
  int powers_of_ten = 1;
  for(int i = 0; i < num_of_digits; i++)
  {
      powers_of_ten *= 10;
  }
  palindrome += (num * powers_of_ten);
  return palindrome;
}
```

بعد از درست کردن پالیندروم ان برگردانده میشود و میتوان نتیجه را دید.
برای توانستن اجرای تمام این فایل های تست که درست کرده ایم باید ان ها را هم در makefile اضافه کنیم تا قابلیت اجرا داشته باشند. در بخش های uprogs , extra

```
_create_palindrome_testing\
_test_move_file\
_test_get_most_invoked\
_test_sort_syscalls\
_test_list_all_processes\
```

```
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\
    printf.c umalloc.c create_palindrome_testing.c test_move_file.c test_get_most_invoked.c test_sort_syscalls.c test_list_all_processes.c\
    README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
    .gdbinit.tmpl gdbutil\
```

و نتیجه را به این شکل میبینیم

```
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Seyyed Mohammad Jazayeri
Seyyed Navid Hashemi
Sobhan Kooshki Jahromi
$ create_palindrome_testing 12
create_palindrome is called for 12
KERNEL: sys_create_palindrome(12)
The palindrome of number 12 is: 1221
$
```

پیاده سازی فراخوانی سیستمی انتقال فایل

در این بخش میخواهیم که سیستم کال move_file را پیاده سازی کنیم برای این بخش در تست ما میاییم و اسم فایل مبدا و ادرس مقصد را به سیستم کال move_file میدهیم و انتظار داریم که فایل را از مبدا به مقصد ببرد. پس برنامه سطح کاربر ما به صورت زیر است

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

int main(int argc, char* argv[]){
    if(argc != 3){
        printf(2, "You didn't provide valid arguments!\n");
        exit();
    }
    else{
        int return_value = move_file(argv[1], argv[2]);
        if(return_value == -1){
            printf(2, "moving file was unsuccessful\n");
            exit();
        }
        else{
            printf(1, "moving was successful!\n");
            exit();
        }
    }
}
```

حالا بیاییم و خود سیستم کال move_file را که تعریف کرده ایم بررسی کنیم بخش اصلی کد ما به صرت زیر است این کد در syscall.c است

```c
int written_bytes = 0;
int read_offset = 0;
int write_offset = 0;

ilock(ip_src);
while ((red_bytes = readi(ip_src, buffer, read_offset, sizeof(buffer))) > 0) {
    read_offset += red_bytes;
    if ((written_bytes = writei(ip_dst, buffer, write_offset, red_bytes)) <= 0) {
        iunlock(ip_src);
        iunlock(ip_dst);
        end_op();
        return -1;
    }
    write_offset += written_bytes;
}
iunlock(ip_src);
iunlock(ip_dst);

char src_name[DIRSIZ];
struct inode *ip_src_parent = nameiparent(source, src_name);
if (ip_src_parent == 0) {
    end_op();
    return -1;
}
ilock(ip_src_parent);
if (remove_file(ip_src_parent, src_name) < 0) {
    iunlock(ip_src_parent);
    end_op();
    return -1;
}
iunlock(ip_src_parent);

end_op();
return 0;
```

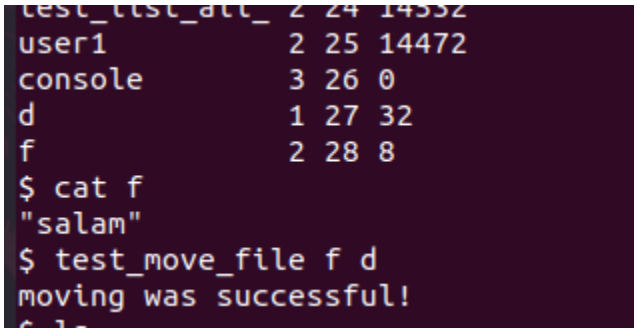
در ابتدا dource و dest را از ارگومان های سیستم کال دریافت میکنیم با استفاده از namei نود مربوط به فایل source را پیدا میکنیم اگر وجود نداشت با خطا خارج میشویم . حالا ما با create_full_pathchar ادرس کامل مقصد را درست میکنیم
در ان ادامه با استفاده از create یک فایل جدید در مقصد ایجاد میکنیم
سپس به صورت قطعه قطعه از فایل مبدا میخوانیم و در فایل مقصد مینویسیم این کار را تا زمانی که تمام اطلاعات منتقل شدند انجام میدهیم سپس باید فایل منبع را حذف کنیم

```
// Helper function to concatenate directory and filename into full path
void create_full_pathchar(char* full_path, char* dir, char* filename) {
    char *p = full_path;
    while (*dir) {
        *p++ = *dir++;
    }
    if (*(p - 1) != '/') {
        *p++ = '/';
    }
    while (*filename) {
        *p++ = *filename++;
    }
    *p = '\0';
}
```

نود پوشه والد فایل منبع را با استفاده از nameiparent پید میکنیم و سپس به تابع remove_file میدهیم تا فایل منبع حذف شود

```
int remove_file(struct inode *dp, char *name) {
    uint i;
    struct dirent de;
    for (i = 0; i < dp->size; i += sizeof(de)) {
        if (readi(dp, (char*)&de, i, sizeof(de)) != sizeof(de)) {
            panic("remove_file: readi");
        }
        if (de.inum == 0) {
            continue;
        }
        if (namecmp(name, de.name) == 0) {
            memset(&de, 0, sizeof(de));
            if (writei(dp, (char*)&de, i, sizeof(de)) != sizeof(de)) {
                panic("remove_file: writei");
            }
            return 0;
        }
    }
    return -1;
}
```

نتیجه این سیستم کال را هم به صورت زیر میتوان مشاهده کرد



پیاده سازی فراخوانی سیستمی مرتب سازی فراخوانی های یک پردازه

برای پیاده سازی این سیستم کال و سیستم کال بعدی ما باید در struct هر process تغییراتی را انجام دهیم در فایل proc.h چند تغییراتی انجام دادیم.

ما با اضافه کردن دو خط اخر میاییم و تعداد system_call هایی که ان process اجرا میکند را نگه داری میکنیم و یک ارایه هم به اندازه تعداد systemcall ها تعریف میکنیم که تعداد استفاده های این process از هر systemcall را ذخیره کنیم و داشته باشیم پس به صورت زیر این دو خط را اضافه میکنیم و در proc.c در خط 114 ما این دو مقدار را initial میکنیم به صفر

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int syscall_count;
  int syscall_history[SYSCALL_NUM];
};
```

```
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
  p->syscall_count = 0;
  memset(p->syscall_history, 0, sizeof(p->syscall_history));
  return p;
}
```

حال در syscall.c میاییم و این اضافه کردن اغداد در count و history را هندل میکنیم به صورت زیر

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

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

در اینجا هر process که اجرا میشود و سیستم کالی را صدا میکند یک واحد syscall_count را اضافه میکنیم و بر اساس شماره سیستم کالی که صدا کرده در ارایه ای که در نظر گرفته ایم عدد ان سیستم کال را اضافه میکنیم

حال تمام امکاناتی که میخواستیم را داریم پس میتوانیم به پیاده سازی sort_syscalls بپردازیم کد این بخش را به صورت زیر میزنیم

```
int sys_sort_syscalls(void){
  int pid;
  if(argint(0, &pid) < 0){
    return -1;
  }

  struct proc* p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if((p->pid == pid) && (p->pid > 0)){
      cprintf("Sorted system calls of %d/frequencies:\n", pid);
      for(int i = 0; i < SYSCALL_NUM; i++)
      {
        if(p->syscall_history[i] > 0)
          cprintf("system call %d : %d\n", i + 1, p->syscall_history[i]);
      }
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

در این سیستم کال ما یک pid میگیریم حالا در ptable که تمام process ها در ان وجود دارد یک حلقه for میزنیم و شماره pid را در ان پیدا میکنیم حالا که این process را پیدا کرده ایم کافی است که فقط ان ارایه history را که اضافه کرده بودیم چاپ کنیم چون ارایه ما به صورت پیش فرض مرتب هست پس دیگر نیاز به یک واحد مرتب سازی نداریم

کد تست خود را هم به صورت زیر میزنیم که در ان سیستم کال را صدا کرده ایم و نتیجه ر هم در عکس بعد از ان میبینید

```
$ test_sort_syscalls 1
Sorted system calls of 1/frequencies:
system call 1 : 1
system call 3 : 1
system call 7 : 1
system call 10 : 2
system call 15 : 2
system call 16 : 87
system call 17 : 1
Sorting compeleted!
$ _
```

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

int main(int argc, char* argv[]){
    if(argc != 2){
        printf(2, "You didn't provide valid arguments!\n");
        exit();
    }
    else{
        int return_value = sort_syscalls(atoi(argv[1]));
        if(return_value == -1){
            printf(2, "Invalid Process ID.\n");
            exit();
        }
        else{
            printf(1, "Sorting compeleted!\n");
            exit();
        }
    }
}
```

پیاده سازی فراخوانی سیستمی برگرداندن بیشترین فراخوانی سیستم برای یک فرآیند خاص

در این بخش هم با کمک متغیر و ارایه ای که در struct هر process اضافه کردیم کارمان را جلو میبریم فقط اینبار سیستم کال get_most_invoked شماره سیستم کالی که بیشترین استفاده را ان process انجام داده است را برگرداند

کد این سیستم کال به صورت زیر است

در این بخش هم ما ابتدا از ptable همان pid را پیدا میکنیم که میخواهیم این سیستم کال بر روی ان اجرا شود سپس در یک for بر روی همان history که خودمان درست کرده ایم بیشترین عدد را پیدا میکنیم و شماره سیستم کالی که بیشترین استفاده را داشته است برمیگردانیم

در عکس های بعدی کد تست خود و نتیجه را میتوانید ببینید

```c
int sys_get_most_invoked_syscall(void){
  int pid;
  if(argint(0, &pid) < 0){
    return -1;
  }

  struct proc* p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      int max = p->syscall_history[0];
      int index = 1;
      for(int i = 0; i < SYSCALL_NUM; i++){
        if(p->syscall_history[i] > max){
          max = p->syscall_history[i];
          index = i + 1;//for array index
        }
      }
      release(&ptable.lock);
      return index;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

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

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

int main(int argc, char* argv[]){
    if(argc != 2){
        printf(2, "Invalid arguments\n");
        exit();
    }
    else{
        int pid = atoi(argv[1]);
        printf(1, "get_most_invoked is called for %d \n", pid);
        int result = get_most_invoked_syscall(pid);
        if(result == -1)
        {
            printf(2, "Invalid process ID.\n");
        }
        else
        {
            printf(1, "The most invoked system call was %d\n", result);
        }

        exit();
    }
}
```

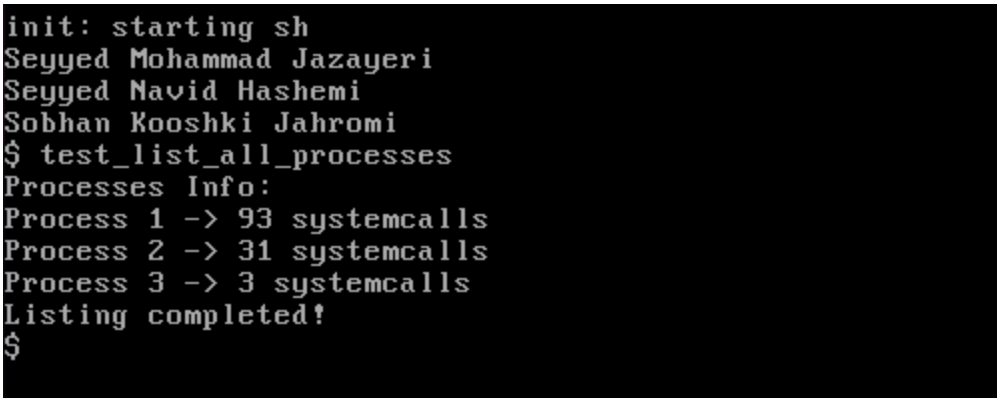
```
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logs
t 58
init: starting sh
Seyyed Mohammad Jazayeri
Seyyed Navid Hashemi
Sobhan Kooshki Jahromi
$ test_get_most_invoked 2
get_most_invoked is called for 2
The most invoked system call was 5
$ _
```

پیاده سازی فراخوانی سیستمی لیست کردن پردازه ها

در این سیستم کال هم با کمک تغییرات در struct هر process که گفته ایم میاییم و لیست تمام process هایی که در حال اجرا هستند را با همان متغیر count چاپ میکنیم
در پیاده سازی این سیستم کال با for زدن بر روی ptable که تمام process ها در ان است انجام میدهیم و برای هر process که در ptable فعال باشد یا همان بزرگتر از صفر باشد متغیر count را هم چاپ میکنیم
پیاده سازی تست و نتیجه را در عکس های زیر میبینید

```
void sys_list_all_processes(void){
  struct proc* p;
  acquire(&ptable.lock);
  cprintf("Processes Info:\n");
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid > 0)
      cprintf("Process %d -> %d systemcalls\n", p->pid, p->syscall_count);
    else
    {
      release(&ptable.lock);
      break;
    }
  }
}
```

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

int main(int argc, char* argv[]){
    if(argc != 1){
        printf(2, "Invalid arguments!\n");
        exit();
    }
    else{
        list_all_processes();
        printf(1, "Listing completed!\n");
        exit();
    }
}
```


init: starting sh
Seyyed Mohammad Jazayeri
Seyyed Navid Hashemi
Sobhan Kooshki Jahromi
$ test_list_all_processes
Processes Info:
Process 1 -> 93 systemcalls
Process 2 -> 31 systemcalls
Process 3 -> 3 systemcalls
Listing completed!
$