به نام خدا

گزارشکار آزمایش دو آزمایشگاه سیستم عامل

اعضای گروه:

علی زمانی 810101436

برنا فروهری 810101480

نیما خنجری 810101416

پرسش 1: کتابخانه های سطح کاربر در xv6 برای ایجاد ارتباط میان برنامه های کاربر و کرنل به کار می روند 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 فایلهای 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 سیستم‌عامل xv6، در مسیرULIB کتابخانه های سطح کاربری را داریم که شامل توابعی هستند که به منظور ایجاد ارتباط بین برنامه های کاربر و کرنل به وجود آمده اند. این کتابخانه‌ها با استفاده از فراخوانی‌های سیستمی به منابع سیستم دسترسی پیدا کرده و امکان بهره مندی برنامه های سطح کاربر از امکانات سیستم عامل و منابع سخت افزاری و نرم افزاری را ممکن می سازند. درxv6 ، این کتابخانه‌ها شامل توابع اساسی هستند که نیاز به دسترسی به سخت‌افزار و منابع سیستم دارند، اما به دلایل **امنیتی و ایمنی**، نمی‌توانند به طور مستقیم به کرنل دسترسی داشته باشند.پس توابعی که در مسیر ULIB می باشند دستوراتی مانند open, read, write و ... را از سطح کاربر به کرنل منتقل می کنند.

هر یک از این توابع یک شماره فراخوانی سیستمی مخصوص به خود دارند. که در فایل syscall.h نعریف شده اند. با استفاده از این شماره ها عمل شناسایی هر فراخوانی سیستم هنگام ارتباط با کرنل به سادگی انجام می شود.

هنگامی که تابعی در ULIB، مثلاً read، فراخوانی می‌شود، این تابع به کمک شماره فراخوانی سیستمی و پارامترهای ورودی، دستورالعملی را اجرا می‌کند که کنترل را از فضای کاربر به فضای کرنل منتقل می‌کند. در معماری‌های x86 این کار با یک دستورالعمل ویژه مثل int یا trap صورت می‌گیرد.

درxv6 ، trap باعث ایجاد وقفه (Interrupt) می‌شود، که CPU را از حالت کاربر به حالت کرنل می‌برد. در این حالت، شماره فراخوانی سیستمی و پارامترهای مربوطه به کرنل منتقل می‌شوند، و کرنل با بررسی شماره فراخوانی، تابع مناسب را اجرا می‌کند.

بعد از دریافت شدن درخواست توسط کرنل و انتقال پارامتر های لازم به آن ، عملیات مربوطه (مانند خواندن فایل، نوشتن در حافظه، یا ایجاد فرآیند جدید) انجام میشود و کرنل نتیجه را از طریق رجیسترهایCPU به فضای کاربر بازمی‌گرداند.

دلایل استفاده از فراخوانی‌های سیستمی و تأثیر آن‌ها:

1. **امنیت و محافظت از سیستم** :فراخوانی‌های سیستمی ارتباط میان کاربر و کرنل را کنترل‌شده و امن می‌کنند. بدون آن‌ها، کاربران ممکن است مستقیماً به سخت‌افزار و داده‌های حساس دسترسی پیدا کنند.

2. **مدیریت منابع** :بسیاری از منابع مانند حافظه و فایل‌ها نیاز به **مدیریت مرکزی** دارند. با استفاده از فراخوانی‌های سیستمی، سیستم‌عامل می‌تواند دسترسی به منابع محدود و مدیریت‌شده‌ای را فراهم کند.

3. **افزایش قابلیت حمل** : به طور کلی قابلیت حمل (Portability) به این معناست که برنامه‌ها و کدهای نوشته‌شده بتوانند بدون نیاز به تغییرات زیاد بر روی سیستم‌عامل‌ها و معماری‌های مختلف اجرا شوند. در سیستم‌عامل‌هایی مانندxv6 ، استفاده از **فراخوانی‌های سیستمی** به افزایش قابلیت حمل کمک می‌کند. با انتزاع سطح پایین سخت‌افزار و منابع، برنامه‌ها به جای دسترسی مستقیم به سخت‌افزار از طریق توابع استاندارد سیستم‌عامل (مانند ..,read, write) با سیستم تعامل می‌کنند. دین ترتیب، برنامه‌های کاربر وابسته به جزئیات خاص سخت‌افزاری یا سیستم‌عامل نبوده و می‌توانند بر روی سیستم‌عامل‌های مختلفی که از استانداردهای مشابهی پیروی می‌کنند، به راحتی اجرا شوند.

در نتیجه هرچند که فراخوانی‌های سیستمی معمولاً به **زمان پردازش بیشتری** نیاز دارند، زیرا باید از فضای کاربر به فضای کرنل منتقل شوند، اما این اندکی کاهش عملکرد در مقایسه با امنیت و مدیریت منابع مناسب است.

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

1. سیستم‌فایل‌های **proc/** و **sys/** : سیستم‌فایل‌های proc/و sys/به عنوان یک واسط مجازی بین کرنل و فضای کاربر عمل می‌کنند proc/ .اطلاعات مربوط به وضعیت سیستم و فرآیندها را در قالب فایل‌های متنی در دسترس قرار می‌دهد، و sys/برای تنظیم و مشاهده پارامترهای کرنل و سخت‌افزار مورد استفاده قرار می‌گیرد .کاربران می‌توانند با خواندن یا نوشتن در این فایل‌ها (مانند فایل‌های معمولی) با کرنل تعامل کنند.

2. سوکت‌ها(Sockets) : سوکت‌ها یکی از اصلی‌ترین روش‌های ارتباط میان‌پردازشی در لینوکس هستند که برای انتقال داده‌ها بین فرآیندها، چه در یک سیستم واحد و چه در شبکه، به کار می‌روند.

3. Shared Memory: shared memory یکی دیگر از روش‌های IPC در لینوکس است که به فرآیندهای مختلف امکان می‌دهد تا از یک فضای حافظه مشترک استفاده کنند. این روش به ویژه برای انتقال سریع داده‌ها بین فرآیندهای کاربر مناسب است، زیرا نیاز به انتقال داده‌ها از طریق کرنل به طور کامل حذف نمی‌شود.

4. Device Files : فایل‌های دستگاه که در مسیر dev/قرار دارند، دسترسی به دستگاه‌های سخت‌افزاری مانند دیسک‌ها، پرینترها و پورت‌های سریال را فراهم می‌کنند. این فایل‌ها به کمک درایورهای کرنل مدیریت می‌شوند و برنامه‌های کاربر می‌توانند از طریق عملیات‌های معمول فایل مانند read و write با این دستگاه‌ها تعامل داشته باشند. این روش به برنامه‌ها اجازه می‌دهد تا بدون نیاز به دسترسی مستقیم به سخت‌افزار، از طریق درایورهای کرنل به دستگاه‌ها دسترسی پیدا کنند.

ioctl.5 : ioctl یک مکانیزم خاص برای ارسال دستورات کنترلی به فایل‌های دستگاه است. این سیستم برای انجام عملیات‌های خاصی که فراتر از read و write هستند، مانند تنظیم پارامترهای دستگاه یا دریافت وضعیت آن، به کار می‌رود . این روش معمولاً برای دسترسی به ویژگی‌های پیشرفته تر دستگاه‌ها استفاده می‌شود.

پرسش 3: آیا باقی تله ها را نمیتوان با سطح دسترسی DPLUSER فعال نمود؟ چرا؟

بیشتر تله‌ها در xv6 و سایر سیستم‌عامل‌ها را نمی‌توان با سطح دسترسی DPL_USER فعال کرد، زیرا این کار می‌تواند مشکلات امنیتی و پایداری را ایجاد کند.

تله‌ها و استثناها اغلب برای مدیریت شرایط بحرانی و غیرعادی (مانند تقسیم بر صفر، خطای دسترسی به حافظه، یا دسترسی غیرمجاز) توسط کرنل ایجاد می‌شوند و کنترل مستقیم آن‌ها توسط کد کاربر می‌تواند باعث عدم امنیت و بی‌ثباتی سیستم شود .اگر اجازه دسترسی به تمامی تله‌ها داده شود، برنامه‌های کاربر می‌توانند مستقیماً این تله‌ها را فراخوانی کنند و ممکن است به داده‌ها و ساختارهای حساس کرنل دسترسی یابند، که می‌تواند منجر به خراب شدن سیستم، سرریز بافرها و بهره‌برداری‌های امنیتی شود . ولی در مقابل، فراخوانی سیستمی به طور خاص برای برقراری ارتباط امن بین کدهای کاربر و کرنل طراحی شده است و بنابراین، تنها فراخوانی سیستمی دارای سطح دسترسی DPL_USER است تا کاربران بتوانند از آن برای درخواست سرویس‌های مجاز از کرنل استفاده کنند.

پرسش 4: در صورت تغییر سطح دسترسی ss و esp روی پشته Push میشود. در غیر اینصورت Push نمیشود. چرا؟

در هنگام فراخوانی یک تله یا سیستم کال، پردازنده باید از سطح دسترسی کاربر (که معمولاً DPL_USERاست) به سطح دسترسی هسته (که DPL_KERNELاست) تغییر کند.

برای اینکه پس از اجرای کد در سطح هسته، پردازنده بتواند به درستی به وضعیت قبلی خود (سطح دسترسی کاربر) بازگردد، لازم است که مقدار ss و esp که به پشته مربوط به سطح کاربر اشاره می‌کنند، ذخیره شود. این مقادیر در پشته ذخیره می‌شوند تا پس از اتمام عملیات کرنل، پردازنده بتواند به درستی به فضای کاربر بازگردد و پشته را به همان شکل اولیه بازیابی کند.

در نتیجه در هنگام تغییر سطح دسترسی از کاربر به کرنل، مقادیر ss و esp به عنوان بخش‌های مهم پشته سطح کاربر ذخیره می‌شوند تا پس از اجرای کد هسته، پردازنده بتواند از پشته سطح کاربر بازگشت کند. اما در صورتی که تغییر سطح دسترسی در تله رخ ندهد (مثلاً اگر تله‌ای که در آن رخ می‌دهد مربوط به کدهای سطح کاربر باشد یا **هیچ تغییر سطحی** در آن نیازی نباشد)، نیازی به ذخیره

اطلاعاتss و esp نیست، زیرا این مقادیر در حال حاضر معتبرند و تغییر نکرده‌اند. در این حالت، پردازنده می‌تواند بدون ذخیره این اطلاعات، عملیات خود را ادامه دهد.

پرسش 5: در مورد توابع دسترسی به پارامترهای فراخوانی سیستمی به طور مختصر توضیح دهید. چرادر ()argptr بازه آدرس ها بررسی میگردد؟ تجاوز از بازه معتبر چه مشکل امنیتی ایجاد میکند؟ در صورت عدم بررسی بازها در این تابع مثالی بزنید که در آن فراخوانی سیستمی ()read_sys اجرای سیستم را با مشکل روبرو سازد.

توابع دسترسی به پارامترهای فراخوانی سیستمی در سیستم‌عامل‌هایی مانند xv6برای استخراج و دسترسی به داده‌های ارسال شده از طرف برنامه‌های سطح کاربر به کرنل طراحی شده‌اند. این توابع معمولاً به پارامترهایی مانند مقادیر عددی یا اشاره‌گرهای حافظه مربوط می‌شوند که در هنگام فراخوانی یک سیستم کال به کرنل ارسال می‌شوند. توابعی مانند argintبرای دریافت پارامترهای عددی و argptrبرای دریافت پارامترهای اشاره‌گر به داده‌ها استفاده می‌شوند. این توابع اطمینان حاصل می‌کنند که پارامترهای ورودی در محدوده‌های معتبر حافظه قرار دارند و از دسترسی‌های غیرمجاز به حافظه جلوگیری می‌کنند.

تابع argptrبرای دسترسی به پارامترهای فراخوانی سیستمی که به صورت اشاره‌گر (pointer) به داده‌ها ارسال شده‌اند، طراحی شده است. وظیفه آن این است که از یک پارامتر ورودی (که معمولاً یک اشاره‌گر به داده‌ها است) اطلاعات مورد نیاز را استخراج کند.

در تابع argptr ابتدا بررسی می‌شود که آدرس‌های داده‌های ورودی به یک بازه معتبر در حافظه اشاره دارند. این کار برای جلوگیری از دسترسی‌های غیرمجاز و خطرناک به حافظه انجام می‌شود. از آنجایی که کاربران نمی‌توانند به صورت مستقیم به حافظه کرنل دسترسی داشته باشند، لازم است که کرنل مطمئن شود پارامترهایی که از سوی برنامه کاربر ارسال می‌شوند، در محدوده مجاز و معتبر قرار دارند.

اگر آدرس‌ها به نواحی خارج از فضای حافظه مجاز کاربر اشاره کنند، این ممکن است به حملات امنیتی مانند buffer overflowیا دسترسی به داده‌های حساس کرنل منجر شود.

از طرفی دسترسی به حافظه غیرمجاز می‌تواند باعث کرش سیستم یا خرابی اطلاعات حساس شود.

مثالی از به مشکل خوردن سیستم با فراخوانی سیستم کال ()read_sys:

فرض کنید برنامه کاربر یک درخواست فراخوانی سیستمی read_sys را برای خواندن داده‌ها از یک فایل به کرنل ارسال می‌کند. در این درخواست، پارامترهای fd (شماره فایل) و buf (آدرس بافر برای ذخیره داده‌ها) به کرنل فرستاده می‌شود.

اگر کاربر آدرسی به عنوان پارامتر buf ارسال کند که به آدرسی از حافظه اشاره می‌کند که غیر مجاز است، یعنی حافظه ای که باید توسط کرنل محافظت شود، در صورت عدم بررسی این آدرس توسط

argptr ، کرنل بدون بررسی صحت این آدرس، داده‌ها را به این آدرس می‌خواند. در نتیجه این عمل، کاربر به داده های حساس کرنل دست پیدا خواهد کرد و این طبیعتا باعث از کنترل خارج شدن حافظه خواهد شد. در ابعاد بزرگتر، این امر می تواند موجب اجرای کد مخرب یا کلاهبرداری اطلاعات شود.

GDB:

1. Make clean
2. Make qemu-gdb
3. باز کردن ترمینال جدید
4. اجرای gdb
5. اضافه کردن breakpoint برای تابع syscall در فایل syscall.c
6. اضافه کردن فایل gdb_test.c :

gdb_test.c

C: > Users > ASUS > Downloads > C gdb_test.c

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

int main() {
    int pid = getpid();
    printf(1, "Process ID: %d\n", pid);
    exit();
}
```

7. اجرای دستور bt : پس از صدا زده شدن این دستور محتویات stack call نمایش داده می شود. stack Call در واقع یک stack است که برای ذخیره سازی سیر اجرای برنامه برای اینکه بتوانیم سیر اجرای درست برنامه را پیگیری کنیم ایجاد میشود. در واقع سیر توابع صدا زده شده را در خود دارد: با اجرای دستور int 64 ، وارد vector64 میشویم که بعد از push شدن

مقدار آن در فایل vector.s به alltraps پرش می کنیم. Alltraps ابتدا frame trap را می سازد و آن را در استک push می کند.

```
(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:141
141       struct proc *curproc = myproc();
(gdb) bt
#0  syscall () at syscall.c:141
#1  0x80105b9d in trap (tf=0x8dffffb4) at trap.c:43
#2  0x8010594f in alltraps () at trapasm.S:20
(gdb)
```

8. عملیات بعدی که باید در این بخش انجام شود این است که دستور down را در بخش خط فرمان gdb اجرا کنیم. این دستور ما را به frame stack بالا تر در bt می برد در واقع ما را تابعی که دیرترصدا زده شده یا همان callee می برد. که در اینجا چون syscall تابعی را صدا نزده است پس callstack دارای هیچ frame stack در این جهت نیست.

```
(gdb) down
Bottom (innermost) frame selected; you cannot go down.
(gdb)
```

9. دستور بعدی دستور up است که دقیقا مخالف دستور down است و اینجا چون به یک دستور عقب ترمی رویم به دستور trap می رسیم.

```
(gdb) up
#1  0x80105b9d in trap (tf=0x8dffffb4) at trap.c:43
43          syscall();
(gdb)
```

10. محتویات رجیستر eax به شرح زیر می باشد:

```
(gdb) print myproc()->tf->eax
$1 = 7
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:141
141       struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$2 = 15
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:141
141       struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$3 = 10
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:141
141       struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$4 = 10
(gdb) c
```

```
(gdb) print myproc()->tf->eax
$18 = 16
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:141
141       struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$19 = 16
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:141
141       struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$20 = 16
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:141
141       struct proc *curproc = myproc();
(gdb) print myproc()->tf->eax
$21 = 1
(gdb) print myproc()->tf->eax
```

Syscall name	7	15	10	16	1
Syscall code	exec	open	dup	write	fork

11. اجرا کردن فایل تست gdb:

```
$ gdb_test
Process ID: 4
$ _
```

Create_palindrome:

برای ساختن سیستم کال create_palindrome لازم است در بخش های مختلفی از سورس کد xv6 تغییراتی را ایجاد کنیم:

1.سیستم کال جدیدی با شماره 22 را به syscall.h اضافه می کنیم:

```
#define SYS_create_palindrome 22
```

2.حالا دکلریشن create_palindrome را به user.h اضافه می کنیم:

```
void create_palindrome(int num);
```

3.حالا در فایل sysproc.c تابعی می سازیم به نام sys_create_palindrome. در این تابع تمام عملیات بازیابی آرگومان ها از رجیستر ها و درست کردن پالیندروم عدد مورد نظر و در نهایت چاپ آن در سطح هسته صورت میگیرد:

```c
#include "syscall.h"
#include "traps.h"


int create_palindrome_num(int num) {
    char str[20];  // Buffer to hold the original number as a string
    //(20 digits to handle large integers)
    int length = 0;

    // Converting our integer to string
    int temp = num;
    while (temp > 0) {
        str[length++] = (temp % 10) + '0';
        temp /= 10;
    }
    str[length] = '\0';

    char palindrome_str[40];  // 2x length buffer to handle the palindrome
    int i, j;
    for (i = 0; i < length; i++) {
        palindrome_str[i] = str[length - i - 1];  // Copying the reversed part
    }
    for (j = 0; j < length; j++) {
        palindrome_str[i++] = str[j];  // Copying the original part
    }
    palindrome_str[i] = '\0';

    cprintf("%s\n", palindrome_str);

    return 0;
}

int sys_create_palindrome(void) {
    int num;

    // Receive the integer argument from the REGISTERS
    if (argint(0, &num) < 0)
        return -1;

    // Generate and print the palindrome in kernel level
    create_palindrome_num(num);

    return 0;
}
```

عملیات بازیابی عدد مورد نظر که آرگومان تابعمان هم هست به وسیله argint(0 , &num) صورت می گیرد.argint عدد صحیح موجود در رجیستر EBX را بازیابی می کند.این گونه سیستم کال ما از رجیستر برای یافتن آرگومان ها و بازیابی آن ها کمک میگیرد.

در تابع create_palindrome_num ، str بافری است که عدد اصلی را نگه می دارد(به صورت کارکتر).

یک بافر دیگر هم داریم که سایز آن دو برابر سایز بافر str در نظر گرفته شده چون قرار است outputنهایی که همان فرم پالیندروم شده عدد اولیه است در آن ذخیره شود. در این بافر به ترتیب کارکترهای عدد اصلی و سپس عدد reverse شده را میریزیم و در نهایت عدد پالندروم نهایی ذخیره شده در بافر palindrome_str را در سطح کرنل چاپ می کنیم.

4. اکنون باید به فایل syscall.c ، sys_create_palindrome را اضافه کنیم:

ابتدا extern int sys_create_palindrome(void) را برای دکلر کردن تابع سیستم کال موردنظر اضافه میکنیم.

سپس، sys_create_palindrome را به syscalls array اضافه می کنیم:

```
extern int sys_create_palindrome(void);
```

```
[SYS_create_palindrome] sys_create_palindrome,
```

5.حال در usys.S ، entry را برای سیستم کالمان اضافه میکنیم:

```
#include "syscall.h"
#include "traps.h"

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

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(create_palindrome)
```

6. حالا که سیستم کال را ساختیم و فایل های مورد نیاز را آپدیت کردیم، کافیست که برنامه ای در سطح یوزر بنویسیم که به وسیله آن بتوانیم درستی کد خود و کارکرد سیستم کالمان را بسنجیم:

کد سی که برای تست کردن سیستم کال create_palindrome به کار میبریم، test_palindrome.c نام دارد:

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

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf(1, "No number provided in order to show the palindrome form of it!\n");
        exit();
    }

    int num = atoi(argv[1]);

    create_palindrome(num);

    exit();
}
```

برای کامپایل کردن این برنامه جدید باxv6 ، باید فایل test_palindrome.c را به لیست برنامه‌های کاربر در فایل Makefile اضافه کنیم.

در Makefile در UPROGS نام این فایل را اضافه می کنیم:

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

حالا xv6 را با دستورات make clean و make ، recompile می کنیم تا شامل برنامه جدید test_palindrome.c بشود.

سپس دستور make qemu را وارد می کنیم تا وارد ترمینال qemu بشویم. در ترمینال qemu ابتدا نام فایل تست (یعنی test_palindrome) را نوشته و با فاصله از آن عدد مورد نظر را وارد میکنیم.

فرم پالیندروم شده عدد (یعنی خودش کانکت شده با معکوسش) به عنوان خروجی نمایش داده می شود.

اگر عددی را هم بعد از تایپ کردن test_palindrome وارد نکنیم، اخطار به ما داده می شود که باید عددی را به عنوان آرگومان تابعمان وارد کنیم.

```
$ test_palindrome
No number provided in order to show the palindrome form of it!
$ test_palindrome 123456789
123456789987654321
$ test_palindrome 120
120021
```

پیاده سازی فراخوانی سیستمی انتقال فایل:

این کد پیاده‌سازی یک تابع سیستمی به نام sys_move_fileدر کرنل است که وظیفه جابه‌جایی یک فایل از یک مسیر به مسیر دیگری (دایرکتوری مقصد) را دارد. این کد به صورت مستقیم با سیستم فایل کار می‌کند و به وسیله‌ی کار با inodeها و ساختارهای دایرکتوری، فایل را از دایرکتوری مبدأ به دایرکتوری مقصد منتقل می‌کند.

تعریف متغیرهای ورودی :ابتدا اشاره‌گرهای src_fileو dest_dirبرای ذخیره‌ی مسیر فایل مبدأ و دایرکتوری مقصد تعریف می‌شوند .به عنوان یک ورودی ساختار دایرکتوری و dirent deoffsetبه عنوان مکان فایل در دایرکتوری مبدأ استفاده می‌شود.

آغاز عملیات سیستم فایل begin_op() :به سیستم می‌گوید که عملیات فایل آغاز شده و آماده استفاده از منابع سیستم فایل است.

پیدا کردن inodeفایل مبدأ :با استفاده از namei(src_file)به دنبال یافتن inodeفایل مبدأ در مسیر src_fileمی‌گردد. اگر فایل مبدأ وجود نداشته باشد، پیام خطا چاپ شده و تابع خاتمه می‌یابد.

مشابه فایل مبدأ، با استفاده از namei(dest_dir)، inodeمربوط به دایرکتوری مقصد یافت می‌شود. اگر دایرکتوری مقصد موجود نباشد، پیام خطا نمایش داده و منابع آزاد می‌شوند.

قفل‌گذاری دایرکتوری مقصد :پس از یافتن inodeدایرکتوری مقصد، قفل آن با ilock(dir_ip) گرفته می‌شود.

قفل‌گذاری فایل مبدأ :پس از یافتن inodeفایل مبدأ، قفل آن با ilock(src_ip)گرفته می‌شود تا از تداخل دسترسی‌های دیگر جلوگیری شود.

ایجاد لینک در دایرکتوری مقصد dirlink(dir_ip, filename, src_ip->inum) :یک لینک جدید برای فایل مبدأ در دایرکتوری مقصد ایجاد می‌کند. این لینک جدید فایل را در دایرکتوری مقصد قابل مشاهده می‌کند. اگر لینک‌دهی موفق نباشد، منابع آزاد شده و تابع خاتمه می‌یابد.

یافتن inodeدایرکتوری والد فایل مبدأ :تابع nameiparent، inodeدایرکتوری والد فایل مبدأ را می‌یابد و در dp_parentذخیره می‌کند. اگر دایرکتوری والد پیدا نشود، منابع آزاد شده و تابع خاتمه می‌یابد.

یافتن inodeفایل در دایرکتوری والد (dirlookup(dp_parent, filename, &offset :فایل را در دایرکتوری والد جستجو کرده و inodeآن را در ipذخیره می‌کند. همچنین، مکان فایل در دایرکتوری (به عنوان offset)ذخیره می‌شود.

حذف ورودی فایل در دایرکتوری والد :ابتدا ساختار deرا با memsetخالی می‌کنیم. سپس با writei، داده‌های deرا در مکان offsetمی‌نویسیم که عملاً ورودی فایل مبدأ را در دایرکتوری والد حذف می‌کند. اگر عملیات موفق نباشد، منابع آزاد می‌شوند و تابع خاتمه می‌یابد.

تغییرات در syscall.h : شماره 23 را به این سیستم کال اختصاص میدهیم.

```
22  #define SYS_close  21
23  #define SYS_move_file 23
24  #define SYS_sort_syscalls 24
```

تغییرات در syscall.c : declaration این تابع را اضافه میکنیم و اضافه کردن سیستم کال به آرایه syscalls

```
134  [SYS_move_file] sys_move_file,
```

تغییرات در makefile :

```
UPROGS=\
    _cat\
    _decode\
    _encode\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _move_file_test\
    _get_process_syscalls\
    _get_syscall_most_invoked\
    _list_active_processes\
```

```
# after running make dist, probably want to
# rename it to rev0 or rev1 or so on and then
# check in that version.

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 move_file_test.c\
    printf.c umalloc.c get_process_syscalls.c get_syscall_most_invoked.c\
    README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
    .gdbinit.tmpl gdbutil\
```

تغییرات در user.h : اضافه کردن prototype تابع سیستم کال

```
int move_file(const char*, const char*);
```

تغییرات در usys.s :

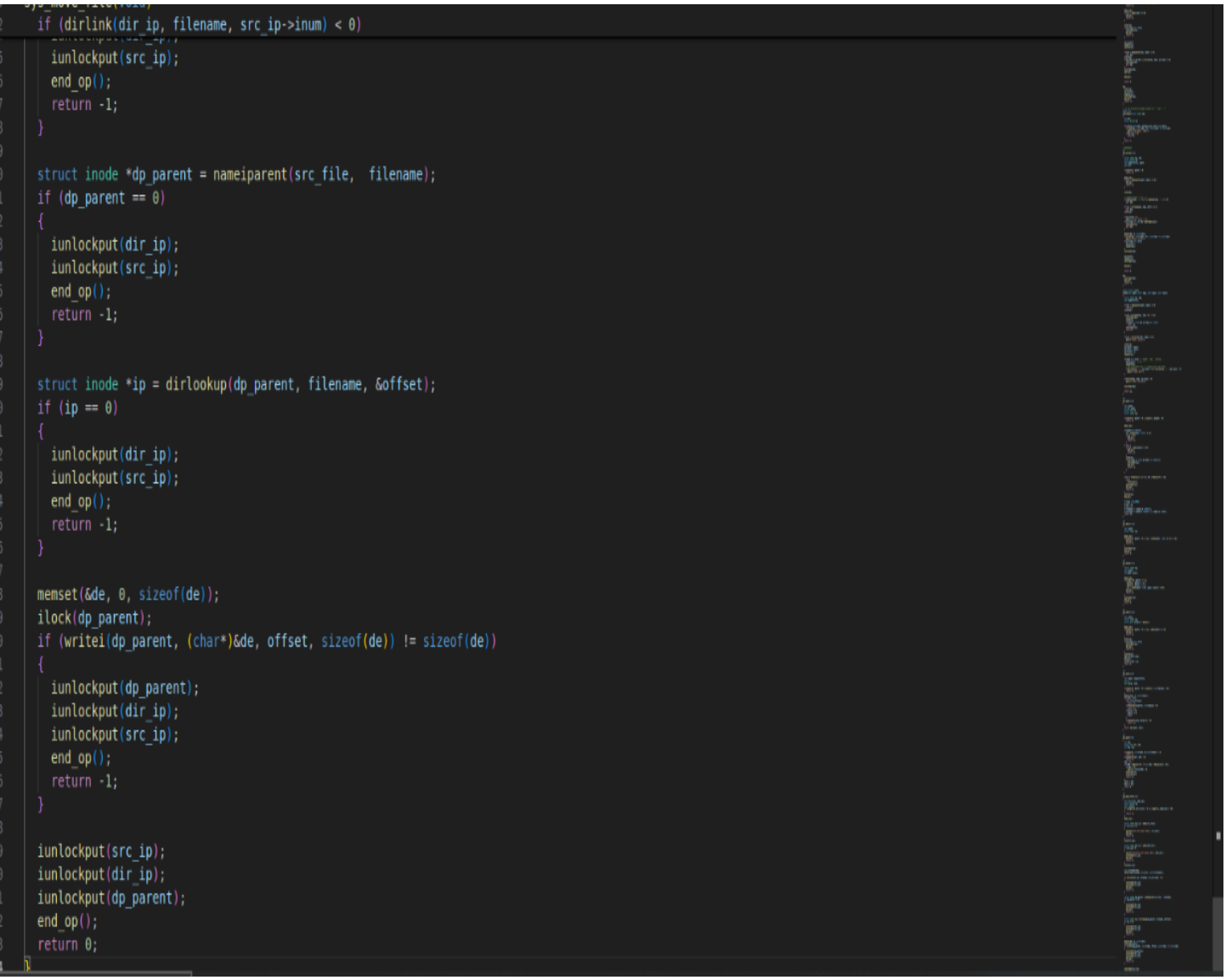
```
SYSCALL(list_active_processes)
SYSCALL(move_file)
```

تغییرات در sysfile.c : سپس سیستم کال را چون مرتبط با کار با فایل است sysfile.c مطابق ذیل تعریف میکنیم

syscall.h | syscall.c | sysfile.c | move_file_test.c | Makefile | user.h | usys.S

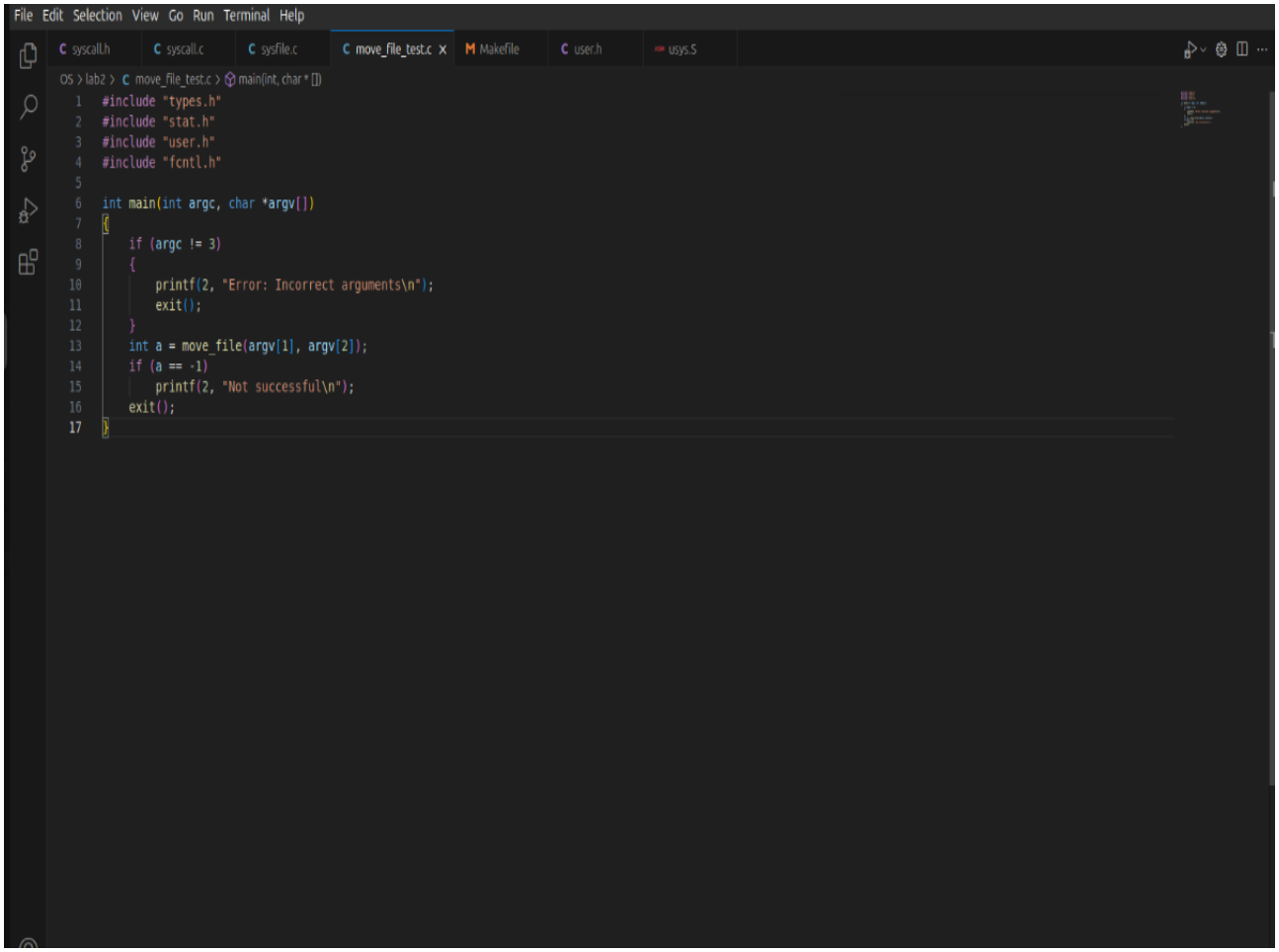
OS > lab2 > sysfile.c > sys_move_file(void)

```
425  sys_pipe(void)
447
448  int
449  sys_move_file(void)
450  {
451    char *src_file, *dest_dir;
452    struct dirent de;
453    uint  offset;
454    if ((argstr(0, &src_file) < 0) || (argstr(1, &dest_dir) < 0))
455    {
456      return -1;
457    }
458    begin_op();
459
460    struct inode *src_ip = namei(src_file);
461    if (src_ip == 0)
462    {
463      cprintf("File not found: %s\n", src_file);
464      end_op();
465      return -1;
466    }
467    ilock(src_ip);
468
469    struct inode *dir_ip = namei(dest_dir);
470    if (dir_ip== 0)
471    {
472      cprintf("Directory not found: %s\n", dest_dir);
473      iunlockput(src_ip);
474      end_op();
475      return -1;
476    }
477    ilock(dir_ip);
478
479    char filename[128];
480    safestrcpy(filename, src_file, sizeof(filename));
481
482    if (dirlink(dir_ip, filename, src_ip->inum) < 0)
483    {
484      iunlockput(dir_ip);
485      iunlockput(src_ip);
486      end_op();
487      return -1;
```



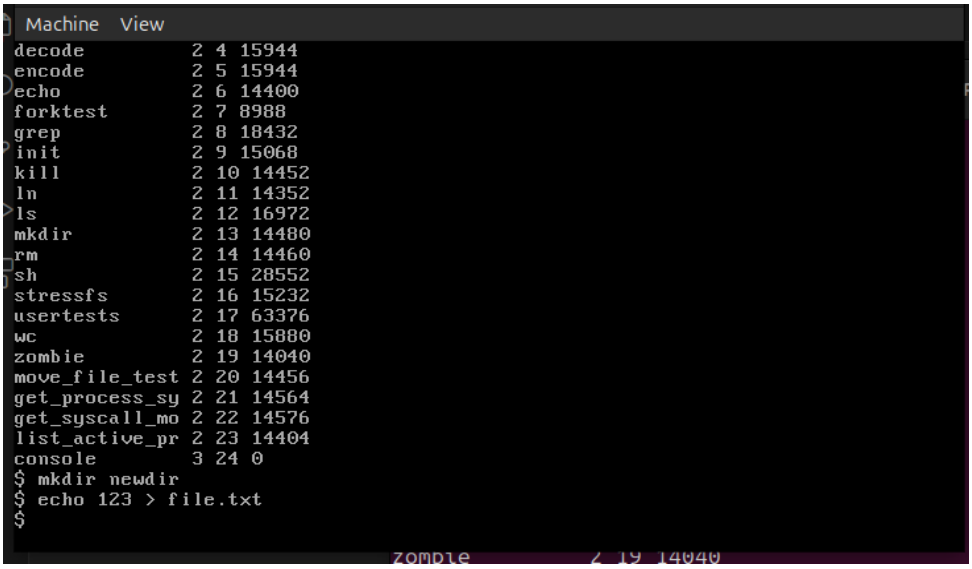
```
  if (dirlink(dir_ip, filename, src_ip->inum) < 0)
    iunlockput(src_ip);
    end_op();
    return -1;
  }

  struct inode *dp_parent = nameiparent(src_file,  filename);
  if (dp_parent == 0)
  {
    iunlockput(dir_ip);
    iunlockput(src_ip);
    end_op();
    return -1;
  }

  struct inode *ip = dirlookup(dp_parent, filename, &offset);
  if (ip == 0)
  {
    iunlockput(dir_ip);
    iunlockput(src_ip);
    end_op();
    return -1;
  }

  memset(&de, 0, sizeof(de));
  ilock(dp_parent);
  if (writei(dp_parent, (char*)&de, offset, sizeof(de)) != sizeof(de))
  {
    iunlockput(dp_parent);
    iunlockput(dir_ip);
    iunlockput(src_ip);
    end_op();
    return -1;
  }

  iunlockput(src_ip);
  iunlockput(dir_ip);
  iunlockput(dp_parent);
  end_op();
  return 0;
}
```

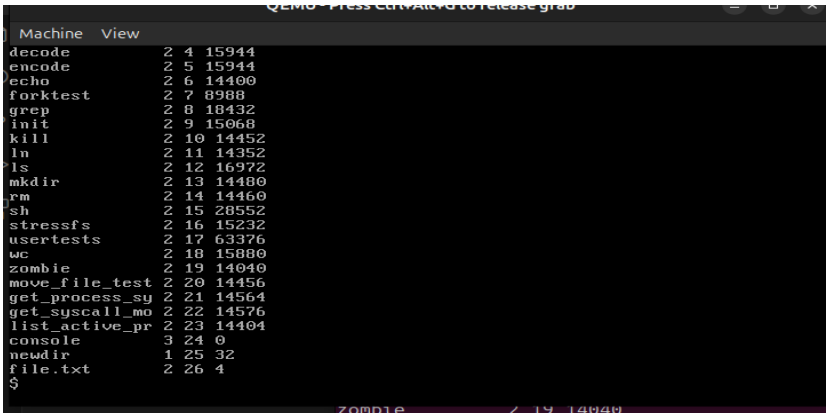
در نهایت فایل زیر را به جهت تست کردن سیستم کال ایجاد شده مینویسیم و آن را در برنامه های سطح کاربر در میک فایل نیز اضافه میکنیم:



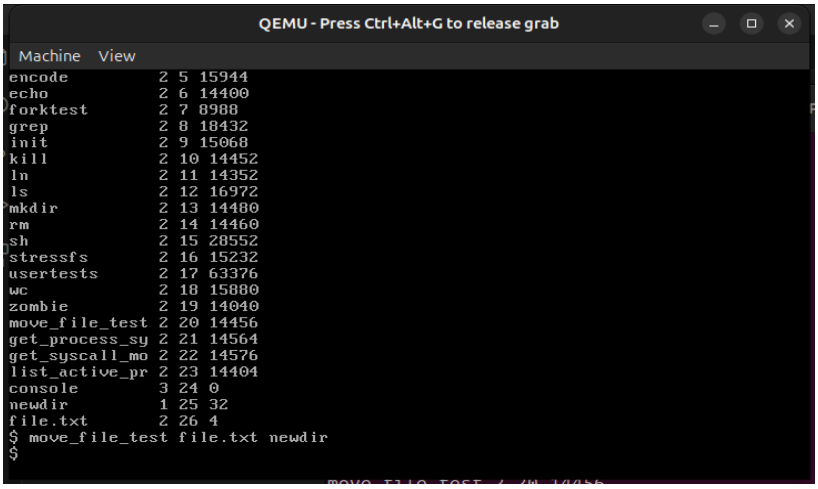
خروجی را به ترتیب:

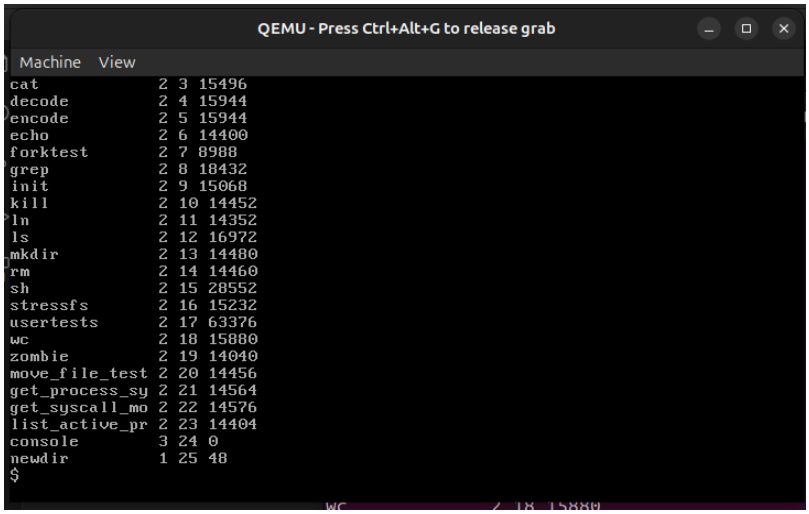
ایجاد دیرکتوری جدید و سپس ایجاد فایل جدید و ریختن مقادیر 1و2و3 در آن



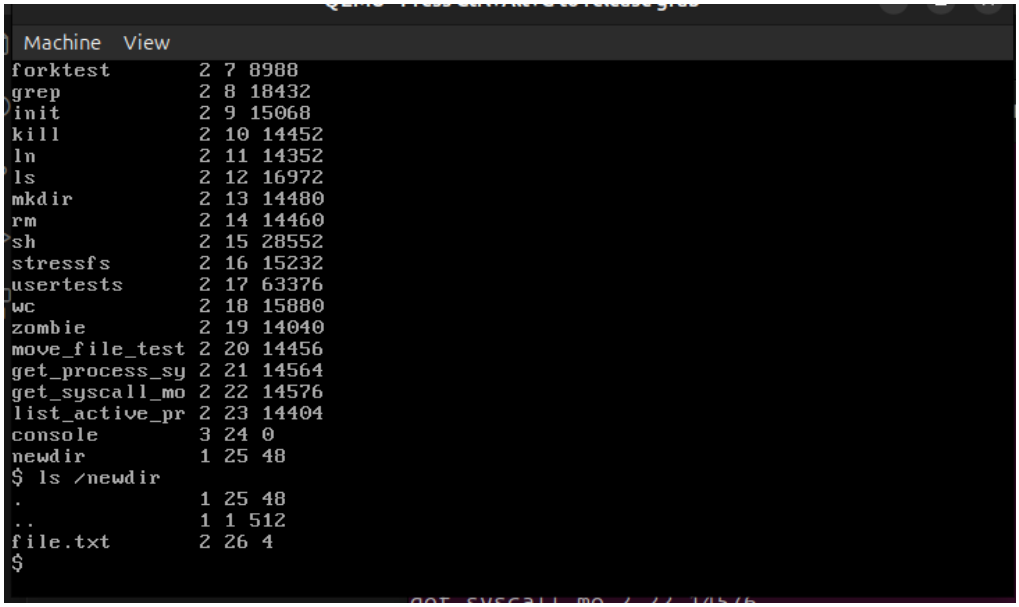


صدا زدن تابع move_file_test :

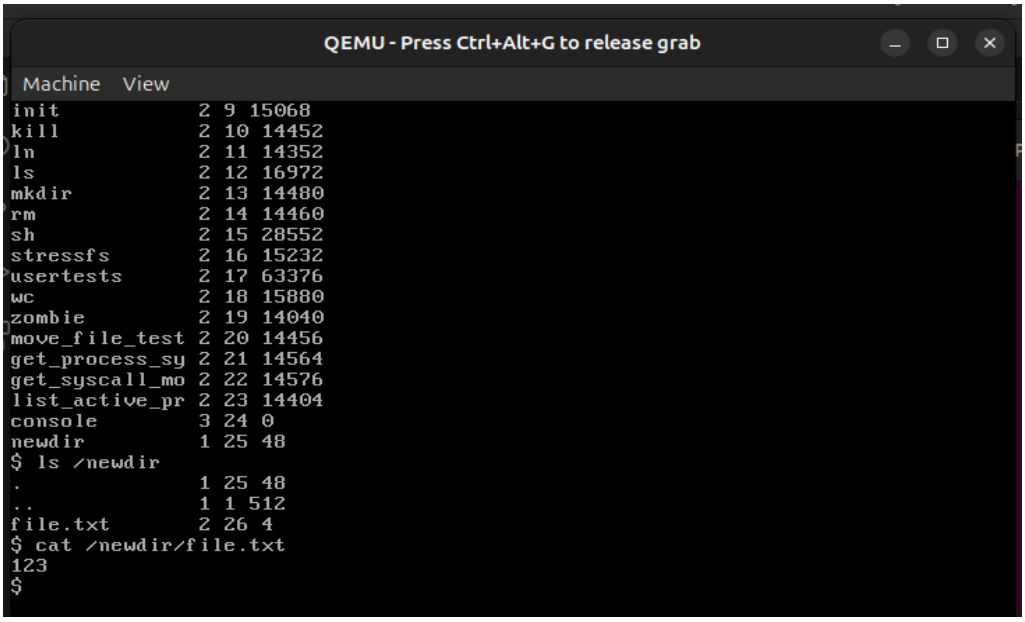




ملاحظه میشود که فایل جا به جا شده و در محل اول هم دیگر وجود ندارد و الان فقط در محل منتقل شده میباشد.



محتویات فایل هم همان باقی مانده است که برای مشاهده از دستور cat استفاده میکنیم.



sort_syscalls():

برای ساختن سیستم کال sort_syscalls لازم است در بخش های مختلفی از سورس کد xv6 تغییراتی را ایجاد کنیم:

1.سیستم کال جدیدی با شماره 24 را به syscall.h اضافه می کنیم:

```
25  #define SYS_sort_syscalls 24
```

2.حالا دکلریشن sort_syscalls را به user.h اضافه می کنیم:

```
26  int sort_syscalls(int pid);
```

3. اضافه کردن prototype تابع سیستم کال درsyscalls.c

```
108   extern int sys_sort_syscalls(void);
```

```
137   [SYS_sort_syscalls] sys_sort_syscalls,
```

4. تعریف تابع درS.usys

```
32    SYSCALL(sort_syscalls)
```

ابتدا ساختار پردازه را به صورتی تغییر می‌دهیم که بتوان فراخوانی‌های سیستمی را ذخیره کنیم.

```
#define MAX_SYSCALLS 27

struct proc {
  int syscalls[MAX_SYSCALLS];
  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[24];               // Process name (debugging)
};
```

```
found:
  for(int i=0; i<MAX_SYSCALLS; i++)
    p->syscalls[i]=0;
```

همچنین برای پیدا کردن پردازه مورد نظر با گرفتن pid می‌توان پردازه مورد نظر را یافت .

```
struct proc* findproc(int pid) {
    struct proc *p;

    // Acquire the process table lock to ensure thread safety.
    acquire(&ptable.lock);

    // Iterate over the process table to find the process with the mat
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->pid == pid) {
            release(&ptable.lock);  // Release the lock before returni
            return p;
        }
    }

    // Release the lock if no process with the given pid is found.
    release(&ptable.lock);
    return 0;
}
```

با هر فراخوانی سیستمی یک واحد به فراخوانی مربوطه اضافه می‌کنیم

```
142
143  void
144  syscall(void)
145  {
146    int num;
147    struct proc *curproc = myproc();
148
149    num = curproc->tf->eax; //define syscall number
150
151    if(num<MAX_SYSCALLS && num>0)
152      curproc->syscalls[num]++;
153
```

هنگام استفاده از فراخوانی سیستمی ساخته شده فراخوانی‌هایی را که حداقل یک بار استفاده شده‌اند را پرینت می‌کنیم

```
153
154  int sys_sort_syscalls(void)
155  {
156    int pid;
157    int counts[MAX_SYSCALLS];
158    if(argint(0, &pid)<0 || argptr(0,(void*)&counts, sizeof(int)*MAX_SYSCALLS<0))
159      return -1;
160
161    struct proc *p = findproc(pid);
162    if(p==0) return -1;
163    for(int i=0; i<MAX_SYSCALLS; i++)
164    {
165      if(p->syscalls[i] != 0)
166        cprintf("%d\n", i);
167
168    }
169    return 0;
170  }
```

برنامه کاربر برای بررسی عملکرد:

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#define MAX_SYSCALLS 27

int main(void) {

    printf(1, "first write system call\n");  // This calls write

    int pid = getpid();
    if (fork() == 0)
    {
        printf(1, "In child process\n");
        exit();
    }
    else
        wait();
    sleep(10);

    if (sort_syscalls(pid) < 0) {
        printf(1, "Error: Failed to retrieve syscall counts\n");
        exit();
    }

    exit();
}
```

:()get_syscall_most_invoked

برای ساختن سیستم کال get_syscall_most_invoked در بخش های مختلفی از سورس کد xv6 تغییراتی را ایجاد کنیم:

1.سیستم کال جدیدی با شماره 25 را به syscall.h اضافه می کنیم:

```
26  #define SYS_get_most_syscalls 25
```

2.حالا دکلریشن get_syscall_most_invoked را به user.h اضافه می کنیم:

```
27  int get_most_syscalls(int pid);
```

3. اضافه کردن prototype تابع سیستم کال درsyscalls.c

```
09  extern int sys_get_most_syscalls(void);
```

```
138  [SYS_get_most_syscalls] sys_get_most_syscalls,
```

4. تعریف تابع درS.usys

```
33  SYSCALL(get_most_syscalls)
```

5. با پیمایش میان تمام فراخوانی‌های سیستمی آنی را که دارای بیشترین تعداد فراخوانی است انتخاب می‌کنیم.

```
int sys_get_most_syscalls(void)
{
  int pid;
  if(argint(0, &pid)<0 , sizeof(int)*MAX_SYSCALLS<0)
    return -1;

  struct proc *p = findproc(pid);
  if(p==0) return -1;
  int syscall_most_invoked = -1;
  for(int i=0; i<MAX_SYSCALLS; i++)
    if(p->syscalls[i] > syscall_most_invoked)
      syscall_most_invoked = i;
  if(syscall_most_invoked<0) return -1;
  cprintf("System call been most invoked: %s - %d times", syscall_names[syscall_most_invoked], p->syscalls[syscall_most_invoked]);
  return 0;
}
```

توجه شود برای پرینت نام فراخوانی سیستمی، یک آرایه برای ذخیره سازی نامهای فراخوانی‌های سیستمی می‌سازیم.

```
const char* syscall_names [] ={
  "none",
  "fork",
  "exit",
  "wait",
  "read",
  "pipe",
  "kill",
  "exec",
  "fstat",
  "chdir",
  "dup",
  "getpid",
  "sbrk",
  "sleep",
  "uptime",
  "open",
  "write",
  "mknod",
  "unlink",
  "link",
  "mkdir",
  "close"
};
```

برنامه کاربر برای بررسی عملکرد:

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#define MAX_SYSCALLS 27

int main(void) {

    printf(1, "first write system call\n");  // This calls write

    int pid = getpid();
    if (fork() == 0)
    {
        printf(1, "In child process\n");
        exit();
    }
    else
        wait();
    sleep(10);

    if (get_most_syscalls(pid) < 0) {
        printf(1, "Error: Failed to retrieve syscall counts\n");
        exit();
    }

    exit();
}
```

:()list_active_processes

برای ساختن سیستم کال list_active_processes لازم است در بخش های مختلفی از سورس کد xv6 تغییراتی را ایجاد کنیم:

1.سیستم کال جدیدی با شماره 24 را به syscall.h اضافه می کنیم:

```
26  #define SYS_get_most_syscalls 25
```

2.حالا دکلریشن list_active_processes را به user.h اضافه می کنیم:

```
28  int list_active_processes(void);
```

3. اضافه کردن prototype تابع سیستم کال درsyscalls.c

```
10  extern int sys_list_active_processes(void);
```

```
139  [SYS_list_active_processes] sys_list_active_processes,
```

4. تعریف تابع درS.usys

```
34  SYSCALL(list_active_processes)
```

برای پیدا کردن پردازه‌های فعال به همراه تعداد فراخوانی‌های سیستمی اجرا شده در آنها پیمایشی در پروسس تیبل انجام می‌دهیم. همچنین برای هر پردازه فعال تعداد فراخوانی‌های سیستمی استفاده شده در آن پردازه را محاسبه می‌کنیم.

```c
int list_active_processes(void) {
    struct proc *p;
    acquire(&ptable.lock);

    cprintf("PID\tName\t\tNumber of syscalls:\n");
    cprintf("---------------------------\n");

    // Iterate over the process table
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->state != UNUSED) {  // Only list active processes
            int num_of_syscalls = 0;
            for(int i=0; i<MAX_SYSCALLS; i++)
              num_of_syscalls+=p->syscalls[i];
            cprintf("%d\t%s\t\t%d\n", p->pid, p->name, num_of_syscalls);
        }
    }

    // Release the process table lock
    release(&ptable.lock);

    return 0;  // Return 0 to indicate success
}
```

```c
201
202
203  int sys_list_active_processes(void) {
204      list_active_processes();
205      return 0;  // Return 0 to indicate success
206  }
207
```

توجه شود برای این امر پردازه‌ای برای لیست کردن پردازه‌های فعال تعریف گردیده است.

برنامه کاربر برای بررسی عملکرد:

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#define MAX_SYSCALLS 27

int main(void) {

    printf(1, "first write system call\n");  // This calls write

    int pid = getpid();
    if (fork() == 0)
    {
        printf(1, "In child process\n");
        exit();
    }
    else
        wait();
    sleep(10);

    list_active_processes();

    exit();
}
```