به نام خدا

دانشگاه تهران

دانشکده برق و کامپیوتر

گزارش کار آزمایشگاه دوم سیستم عامل

پرنا اسدی

مرتضی بهجت

امید پناکاری

فهرست بندی گزارش کار

بخش اول: سوالات پروژه

- **کتابخانه های استفاده شده در xv6 را از منظر استفاده از فراخوانی های سیستمی و علت این استفاده بررسی نمایید**

کتابخانه های سطح کاربر به اسامی ulib.o usys.o printf.o umalloc.o وجود دارند که در هر کدام از فراخوانی های سیستمی مشخصی استفاده شده است.

نام تابع	محل قرارگیری	نام فراخوانی سیستمی	علت استفاده
putc	printf	write	به منظور نوشتن کارکتر ها در فایل مشخص شده استفاده شده است.
morecore	umalloc	sbrk	برای افزایش حافظه ،ابتدا اندازه حافظه پردازه را گرفته، سپس اندازه حافظه اش را به مقدار n افزایش میدهد.
gets	ulib	read	به منظور خواندن کارکتر ها از فایل مشخص شده استفاده شده است.
stat	ulib	open	برای ساختن فایل جدید و یا باز کردن فایل با مسیر مشخص شده(پس از بررسی شرایط) استفاده میشود.
stat	ulib	close	برای بستن فایلی که از قبل باز شده است و آدرس فایل باز شده در پردازه را 0 میکند.

در فایل usys.S تمام سیستم کال ها استفاده شده اند. در این فایل اسمبلی gcc با استفاده از یک C preprocessor استفاده میکند و با توجه به کد داده شده در این قسمت، ابتدا کامند زیر را میسازد(/ ها در این مرحله حذف شده اند):

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

اوپراتور ## توکن سمت راست و چپ خود را با هم concat میکند. به عنوان مثال داریم:

فرض کنیم در رجیستر esx سیستم کال fork قرار داده شده است.

```
.globl fork; fork: movl $SYS_fork, %eax; int $T_SYSCALL; ret
```

در این مرحله ، کامند بالا را ترجمه میکنیم:

- **شناسه تابع fork پابلیک میشود**
- **برچسب frok را تعریف میکنیم(که مانند تابع عمل میکند)**
- **در این تابع:**

1. **مقدار SYS_fork به رجیستر esx اساین میشود.**
2. **یک وقفه به کد T_SYSCALL ساخته میشود.**
3. **این تابع به عنوان مقدار بازگشتی قرار میگیرد.**

- **دقت شود فراخوانی های سیستمی تنها روش دسترسی سطح کاربر به هسته نیست. انواع این روشها را در لینوکس به اختصار توضیح دهید.**

به مجموعه راه های دسترسی کاربر به هسته system API میگویند که در لینوکس شامل system calls ، pseudo-file و تابع های کتابخانه libc میباشد و هرکدام از آنها ممکن است API هایی وجود داشته باشد که بلااستفاده مانده اند.

Pseudo-file : یک فایل سیستم است که درواقع فایل نیست بلکه ورودی های مجازی ای دارد که خود سیستم فایل در محل آنها را میسازد. نمونه ای از این pseudo-file ها ، فایل proc میباشد که در بسیازی از سیستم عامل ها وجود دارد که به طور داینامیک مسیر فایل هایی برای هر پردازه میسازد. به طور کلی pseudo-file دارای اطلاعاتی از سیستم در RAM درحال اجراست.در لینوکس dev/دارای فایل هایی مانند #dev/tty/ که به صورت داینامیک تولید میشوند.

Libc : تابع های C و POSIX در آن پیاده سازی شده است. برای صدا کردن یک سیستم کال از سطح کاربر نمیتوان مستقیم مانند تابع های عادی با آن رفتار کرد.بنابراین باید توابعی برای دسترسی به سیستم کال ها پیاده سازی شود ولی برخی از تابع های پرکاربرد در libc پیاده سازی شده اند. لینوکس با ترکیب هسته و libc این POSIX را پیاده سازی کرده است. لزوما هر POSIX یک فراخوانی سیستمی را صدا نمیکند و اگر هم انجام دهد لزوما هسته این درخواست را تایید نمیکند.

- **آیا باقی تله ها را نمیتوان با سطح دسترسی USER_DPL فعال نمود؟ چرا؟**

نه نمی‌توان چنین کاری را انجام داد زیرا برنامه‌ها سمت کاربر غیر قابل اعتماد هستند و اگر به آن‌ها اجازه دهیم که از دیگر trapها استفاده کنند ممکن است در کار سیستم‌عامل اختلال ایجاد کنند. برای مثال یک برنامه می‌تواند به اشتباه ارور تقسیم بر صفر را گزارش کند.

- **در صورت تغییر سطح دسترسی، ssو esp روی پشته Push میشود. در غیراینصورت Push نمیشود. چرا؟**

رجیستر esp برای نگه داشتن سر استک و ss برای نگه داشتن بلاکی از مموری که استک در آن ذخیره شده استفاده می‌شود. هنگامی که سطح دسترسی در فراخوانی trap تغییر می‌کند این مقادیر در ابتدای استک کرنل push می‌شوند تا بتوان پس از پایان trap به برنامه اصلی بازگشت.

- **در مورد توابع دسترسی به پارامترهای فراخوانی سیستمی به طور مختصر توضیح دهید. چرا در argptr بازه آدرسها بررسی میگردد؟ تجاوز از بازه معتبر، چه مشکل امنیتی ایجاد میکند؟ در صورت عدم بررسی بازهها در این تابع، مثالی بزنید که در آن، فراخوانی سیستمی read_sys اجرای سیستم را با مشکل روبرو سازد.**

در این تابع ورودی که بع تابع داده شده به عنوان یک عدد خوانده شده و از آنجا که میتوان در زبان c از اعداد به عنوان اشاره گر استفاده کرد ممکن ادرس بعد از تبدیل و استفاده از آن به عنوان یک ادرس به خانه های خارج از حافظه پردازه فعلی بشود که در صورت استفاده از این مقدار پردازه دچار segmentation trap شده و پردازه از بین میرود. برای ایجاد این مشکل با سیستم کال sys_read کافی است ارگومان دوم انقد بزرگ داده شود که از فضای پردازه فعلی خارج تر باشد در ادامه هنگامی که میخواد نوشتن روی این ارایه اتفاق بی افتد پردازه کشته میشود.

- **چگونه هنگام بازگشت به سطح کاربر، اجرا از همان خطی که متوقف شده بود، دوباره از سر گرفته میشود؟ فرایند را توضیح دهید.**

بعد از اتمام اجرای systemcall **برای بازگشتن به** userspace **برنامه به بخش** trapret **میرود در این تابع ابتدا** esp **که انتهای استک است به** p->tf **مقدار دهی میشود سپس مقدار رجیستر های** %ds, %es, %fs, %gs **از** stack **پاپ شده و بازیابی میشوند. در ادامه با اجرای دستور** addl **دو مقدار** trapno, errorcode **جا افتاده و دستور** iret **اجرا میشود تا بدین ترتیب رجیستر های** %esp, %flags, %eip, %cs **مقدار دهی و بازیابی شوند. اکنون مود اجرایی به** userspace **ست شده و ادامه دستورات پیگیری میشود.**

- **توضیح دهید عدم توالی داده های یک فایل روی دیسک، چگونه ممکن است در فرایند بازیابی آن پس از حذف مشکل ساز شود.(امتیازی)**

برای بازیابی فایل های حذف شده، سراغ داده هایی میرویم که از دیسک حذف شده اما هنوز روی آنها باز نویسی اتفاق نیافتاده بدین ترتیب ممکن است بتوانیم بخشی از داده های باقی مانده را باز گردانیم برای اگر داده ها پشت هم نباشند برای اینکه یک فایل را تولید کنیم باید قسمت های متفاوت آن را از سراسر دیسک جمع آوری کنیم که با توجه به اینکه صرفا با بایت ها طرف هستیم و اطلاعات مربوط به مکان های نگهداری (metadata) **حذف شده است دچار مشکل میشویم. هر چند باز هم امکان بازیابی فایل ها وجود دارد صرفا سختی آن بیشتر میشود و احتمال بازیابی کاهش میابد.**

بخش دوم: اضافه کردن فراخوانی سیستمی

نحوه‌ی اضافه کردن system call جدید: (با نام name)

۱- ابتدا کد مربوط به system call جدید را با یک macro به نام SYS_name در فایل syscall.h اضافه می‌کنیم.

۲- سپس پیاده سازی مربوط به system call را در sysproc.c یا sysfile.c با نام sys_name انجام می‌دهیم.

۳- سپس کد system call را به پیاده‌سازی آن در syscall.c مپ می کنیم.

۴- عبارت (SYSCALL(name را به فایل usys.S اضافه می‌کنیم.

۵- در نهایت prototype مربوط به آن را به user.h اضافه می‌کنیم تا توسط برنامه‌های سمت کاربر قابل استفاده باشد.

✓ فراخوانی سیستمی calculate_sum_of_digits

برای گرفتن ورودی مربوط به این system call از trapframe مربوط به process کنونی استفاده می‌کنیم.

myproc() -> tf -> ebx

✓ اضافه کردن فراخوانی سیستمی چاپ سکتور ها

1- ابتدا اینترفیس تابع getsectors را در فایل def.h اضافه میکنیم.این تابع برای گرفتن سکتور های موجود استفاده میشود.

2- سپس اینترفیس تابع getfilesectors را به همان فایل اضافه میکنیم.

3- سپس بدنه آنها را در فایل file.c و fs.c اضافه میکنیم.

4- برای افزودن اینترفیس برای فراخوانی سیستمی، در فایل user.h نام getfilesectors را با ارگومان هایش قرار میدهیم.

5- کد این سیستم کال را در فایل syscall.h را 24 قرار میدهیم.

6- در نهایت تابع فراخوانی سیستم را در فایل sysfile.c پیاده سازی میکنیم.

7- فایل مثال آن در fi قرار گرفته و در makefile در قسمت EXTRA و UPROG اضافه میکنیم و میتوانیم در پوسته از آن استفاده کنیم.

✓ اضافه کردن فراخوانی چاپ پدر پردازه

فراخوانی get_parent_id: توضیح خاصی ندارد

myproc() -> parent -> pid

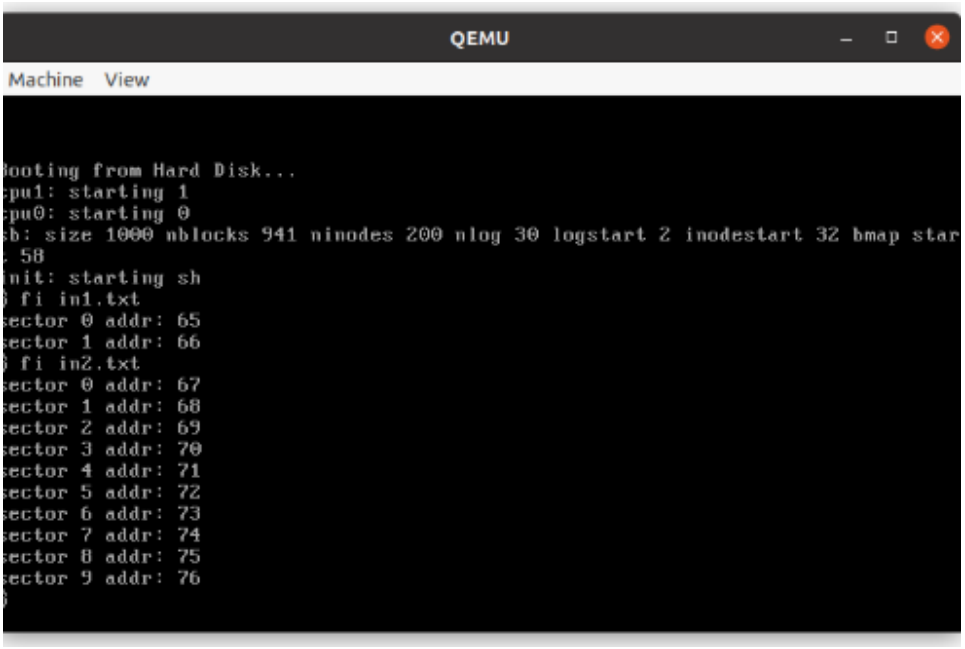
✓ اضافه کردن فراخوانی تعویض پدر پردازه

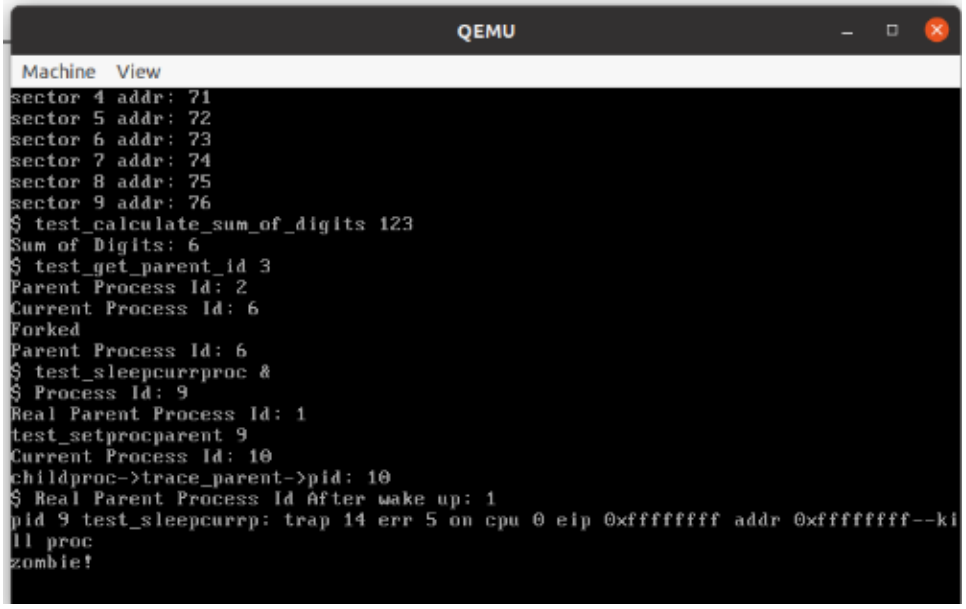
1- **در فایل proc.h به استراکت پردازه یک متغیر جدید به نام** trace_parent **اضافه میکنیم که پوینتری به استراکت پدر جدید است.**

2- **سپس در فایل** proc.c **تابع** getprocbypid **را اضافه میکنیم که با استفاده از** id **استراکت پردازه مد نظر را به ما برگرداند.**

3- **برای مقدار دهی اولیه** trace_parent **را برابر** parent **قرار میدهیم(مشابه لینوکس) و این کار را در تابع** fork **انجام میدهیم.**

4- **در آخر تابع** setprocparent **را در همین فایل پیاده سازی میکنیم.**


QEMU
Machine View
ooting from Hard Disk...
pu1: starting 1
pu0: starting 0
b: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
58
nit: starting sh
fi in1.txt
ector 0 addr: 65
ector 1 addr: 66
fi in2.txt
ector 0 addr: 67
ector 1 addr: 68
ector 2 addr: 69
ector 3 addr: 70
ector 4 addr: 71
ector 5 addr: 72
ector 6 addr: 73
ector 7 addr: 74
ector 8 addr: 75
ector 9 addr: 76


QEMU
Machine View
sector 4 addr: 71
sector 5 addr: 72
sector 6 addr: 73
sector 7 addr: 74
sector 8 addr: 75
sector 9 addr: 76
$ test_calculate_sum_of_digits 123
Sum of Digits: 6
$ test_get_parent_id 3
Parent Process Id: 2
Current Process Id: 6
Forked
Parent Process Id: 6
$ test_sleepcurrproc &
$ Process Id: 9
Real Parent Process Id: 1
test_setprocparent 9
Current Process Id: 10
childproc->trace_parent->pid: 10
$ Real Parent Process Id After wake up: 1
pid 9 test_sleepcurrp: trap 14 err 5 on cpu 0 eip 0xffffffff addr 0xffffffff--ki
ll proc
zombie!

پایان