زیبا امیدوار، نازنین صبری، سمانه راسخ

**گام یک:**

سوال ۱ و ۳: مود های گوناگون برای cpu حالت‌های اجرای اند که در هر یک از آن‌ها محدودیت مشخصی روی دستوراتی که cpu و برخی سخت‌افزارهای خاص می‌تواند اجرا کند قرار داده می‌شود. این نوع طراحی امکان ارائه خدمات اضافه به یک کاربر و یا در یک مود جدید در مقایسه با بقیه را امکان‌پذیر می‌سازد.
در تئوری فقط دستورات اصلی کرنل در حالت کرنل اجرا می‌شود و بقیه‌ی برنامه‌ها با استفاده از سیستم کال‌ها با سیستم‌عامل در ارتباط خواهند بود و دستورات سطح سیستم را اجرا می‌کنند. در عمل چون سیستم کال زمان می‌برد و سربار دارد پس برخی برنامه‌ها که حساسیت به زمان دارند (مثلا device driverها) با دسترسی کرنل مود اجرا می‌شوند.
کرنل مود حالتی است در که در آن کد به تمام زیر ساخت‌های سخت‌افزاری دسترسی کامل دارد، می‌تواند به هر آدرس خانه‌ی حافظه دسترسی داشته باشد و هر دستوری را که می‌خواهد در CPU اجرا کند. در حالت یوزر کد امکان دسترسی مستقیم به سخت‌افزار و حافظه را ندارد. همچنین امکان دسترسی به دستگاه‌های ورودی و خروجی را نداریم و به طور کلی در حالت یوزر دستوراتی که می‌تواند حالت عمومی سیستم را تغییر دهد و روی سایر پروسس‌ها تاثیر بگذارند قابل اجرا نیستند. بخش عمده‌ای از برنامه‌هایی که ما به طور روزانه اجرا می‌کنیم در حالت یوزد هستند. اگر می‌خواستیم دسترسی‌های کرنل را به تمام یوزرها بدهیم امنیت سیستم قابل تامین نبود.

سوال ۲: دسته بندی فراخوانی های سیستمی با توجه به کارکردهایشان:

**Process control**

end, abort
load, execute
create process, terminate process
get process attributes, set process attributes
wait for time
wait event, signal event
allocate and free memory

**File management**

create file, delete file
open, close file
read, write, reposition
get and set file attributes

**Device management**

request device, release device
read, write, reposition
get device attributes, set device attributes
logically attach or detach devices

**Information maintenance**

get time or date, set time or date
get system data, set system data
get and set process, file, or device attributes

**Communications**

create, delete communication connection
send, receive messages
transfer status information
attach and detach remote devices

مثال هایی در ویندوز و یونیکس:

Process control:  unix: fork(), exit(), wait()
Windows: CreateProcess(),ExitProcess(),WaitForSingleObject()
File management:  unix: open(),read(),write(),close()
Windows: CreateFile(),ReadFile(),WriteFile(),CloseHandle()
Device management: unix: ioctl(),read(),write()
Windows: SetConsoleMode(),ReadConsole(),WriteConsole()
Information maintenance: unix: getpid(),alarm(),sleep()
Windows: GetCurrentProcessID(), SetTimer(),Sleep()
Communication:  unix: pipe(),shmget(),mmap()
Windows: CreatePipe(),CreateFileMapping(),MapViewOfFile()
Protection: unix: chmode(), umask(), chown()
Windows: SetFileSecurity(), InitlializeSecurityDescriptor(), SetSecurityGroup()

فراخوانی سیستمی دو بخش دارد:

برای فضای کاربر امکان پذیر نیست که کد kernelرا به طور مستقیم اجرا کند.برنامه های فضای کاربر باید به kernel یک سیگنال ارسال کنند که میخواهند یک system call را اجرا کنند و سپس به حالت کرنل سوییچ کنند.مکانیزم فرستادن سیگنال به کرنل یک intrupt نرم افزاری است.یک exception ایجاد میشود؛سیستم به kernel mode تغییر میکند و exception handler را اجرا میکند. در اینجا exception handler همان system call handler است. این intrupt نرم افزاری در x86، داخل intrupt number 128 است. که از طریق دستور 0x80$ انجام میشود. این مساله باعث سوییچ به کرنل و اجرای exceptio vector 128 که همان system call handler است میشود.وارد شدن به فضای کرنل به تنهایی کافی نیست زیرا فراخوانی های سیستمی چندگانه ای وجود دارند که همه به شیوه ی مشابه وارد کرنل میشوند. بنابراین برای تشخیص نوع فراخوان سیستمی باید system call number به کرنل فرستاده شود. این مقدار در رجیستر eax ذخیره میشود. System call handler مقدار eax را میخواند . تابع system-call() اعتبار suystem call number را میسنجد و در صورت نامعتبر بودن تابع ENO-sys را باز میگرداند. سپس آرگومان های فراخوانی سیستمی از طریق رجیسترهای سخت افزاری دیگر منتقل شده و با اجرای یک دستور intrupt صورت میگیرد.بعد از اجرای intrupt، شماره ی فراخوانی سیستمی به عنوان ایندکس جدولی از اشاره گرها استفاده میشود تا آدرس شروع کد راه انداز فراخوانی سیستمی به دست آید.سپس processبه این آدرس پریده و امتیازهای processاز مد کاربر به مد هسته منتقل می شوند.فهرست شماره های فراخوانی های سیستمی نسخه های اخیر هسته ی لینوکس در مسیر include/asm-x86/unistd.h/ موجود است.(برای نمونه، _NR_close که متناظر با فراخوانی سیستمی close() است برای بستن یک فایل استفاده میشود و با مقدار ۶ تعریف میشود)معمولا فهرست اشاره گرهای راه انداز فراخوانی های سیستمی در فایلِ arch/x86/kernel/entry-64.s زیر سر خط( ENTRY(sys_call_table میشود.

entry_64.S:

```
 * Syscall return path ending with SYSRET (fast path)
  * We must check ti flags with interrupts (or at least preemption)
  * off because we must *never* return to userspace without
  * processing exit work that is enqueued if we're preempted here.
  * In particular, returning to userspace with any of the one-shot
  * flags (TIF_NOTIFY_RESUME, TIF_USER_RETURN_NOTIFY, etc)
 * Has incomplete stack frame and undefined top of stack
 *Reload arg registers from stack in case ptrace changed them.
  *We don't reload %rax because syscall_trace_entry_phase2() returned
  * the value it wants us to use in the table look
* edi:  mask to check *
* edx:  work, edi: workmask
 handle signals and tracing — both require a full stack frame
* Check for syscall exit trace */
```

منابع:

http://www.cs.iit.edu/~cs561/cs450/system_calls/style/9.html http://scanftree.com/operating-system/Types-of-System-Calls

آیا روش های دیگری غیر از فراخوانی سیستمی برای دسترسی به اطلاعات و سرویس های هسته وجود دارد؟
در لینوکس system call ها ابزار هایی هستند که interface بین فضای کاربر و هسته هستند.system call ها تنها ورودی مجاز به هسته هستند به غیر از exception ها و trap ها. در واقع interface های دیگر از قبیل device files یا /proc نهایتا از طریق system call ها قابل دسترسی هستند.یکی از روش های دسترسی به سرویس های کرنل استفاده از API ها هستند.در واقع با صدا زدن یک API یک سیستم کال فراخوانی میشود.علت استفاده از API درگیر نشدن با جزییات و portability بودن برنامه است.یک روش دیگر message passing است.در این روش process ایجاد شده در فضای کاربر یک message ایجاد میکند و سرویس مورد در خواستش را توصیف میکند.بعد از آن از یک تابع مطمین برای فرستادنmessage به یک process مورد اعتماد سیستم عامل استفاده میکند.این تابع مشابه یک trap عمل میکند و message را بررسی کرده و به کرنل مود سوییچ میکند.به همین روش نتیجه ی سرویس در خواستی را به فضای کاربر برمیگرداند.
منبع:کتاب love

سوال ۴: یکی از روش‌ها انتقال پارامترها با استفاده از رجیستر است که در روش تعداد پارامترهایی که می‌توانیم منتقل کنیم محدود است و اینکه دقیقا از کدام رجیستر ها برای انجام این کار می‌توانیم استفاده کنیم به ماشین ما بستگی دارد مثلا:

> On x86-64 UNIX systems, including Linux and default NetRun, the first six parameters go into rdi, rsi, rdx, rcx, r8, and r9.
> On Windows 64, the first four parameters go into rcx, rdx, r8, and r9
> On 32-bit x86 systems, no parameters go in registers, they're all on "the stack" (wait a month to hear about this!).

پس زمانی که می‌خواهیم تعدادی بیشتر از ۶ پارامتر (برای یونیکس مثلا) پاس دهیم به سراغ استک می‌رویم.
https://stackoverflow.com/questions/19858980/how-does-assembly-do-parameter-passing-by-value-reference-pointer-for-differe

سوال ۵: در رجیستر ها ذخیره می شوند.در x86-32 ، رجیستر های ebx, ecx, edx, esi و edi به ترتیب حاوی 5 آرگومان اول می باشند. در صورت بیشتر بودن تعداد آرگومان ها، یک رجیستر برای نگهداری اشاره گر به فضای کاربر نگهداری می شود که در آن فضا، همه ی پارامتر ها ذخیره شده اند.
مقدار بازگشتی هم با رجیستر به فضای کاربر داده می شود.که در x86، در رجیستر eax نوشته می شود.
منبع:بخش parameter passing کتاب love صفحه 74

سوال 6: الف)کرنل دو متد برای بررسی های لازم و کپی کردن از فضای کاربر و کپی کردن به آن تهیه می کند. برای نوشتن در فضای کاربر،متد copy_to_user () استفاده می شود که 3 پارامتر می گیرد. پارامتر اول آدرس حافظه مقصد در فضای آدرس process می باشد و پارامتر دوم، اشاره گر به مبدایی است در فضای کرنل که می خواهیم از آن کپی کنیم، و پارامتر سوم هم سایز داده به بایت می باشد. برای خواندن از فضای کاربر، از متد copy_from_user() استفاده می شود که شبیه copy_to_user() می باشد. این تابع از پارامتر دوم به اندازه ی پارامتر سوم بایت داده میخواند و در پارامتر اول می ریزد.هر دو این تابع ها تعداد بایت هایی را که نتوانسته اند بخوانند را برمی گردانند و در صورت موفقیت صفر برمی گردانند.
ب)حافظه به دو بخش مجزا تقسیم می شود. 1) فضای کاربر که قسمتی که پروسه های کاربر در آن اجرا می شوند.نقش کرنل مدیریت برنامه های در حال اجرا در این فضا می باشد که با هم تداخل نداشته باشند.2) فضای کرنل که فضایی است که کد کرنل در آن ذخیره و اجرا می شود.
پروسه های در حال اجرا در فضای کاربر فقط به بخش محدودی از حافظه دسترسی دارند، در حالی که کرنل به تمام حافظه دسترسی دارد.علاوه بر این، پروسه های فضای کاربر به فضای کرنل دسترسی ندارند و فقط می توانند توسط رابطی که کرنل در اختیار آنها قرار می دهد(system calls) به بخش کوچکی از کرنل دسترسی پیدا کنند.

برنامه های فضای کاربر و کرنل، فضای آدرس های متفاوتی دارند به همین دلیل کپی کردن از یکی به دیگری نیازمند تغییر فضای آدرس میباشد.

https://unix.stackexchange.com/questions/87625/what-is-difference-between-user-space-and-kernel-space

:SYSCALL
یک تابع کتابخانه ای است که system call را فراخوانی میکند که interface زبان اسمبلی آن عدد و آرگومان خاصی دارد.استفاده از syscall مفید است به عنوان مثال برای system callای که wrapper function در کتابخانه ی c ندارد. syscall() رجیسترهای cpu را قبل از ایجاد system call ذخیره میکند .رجیستر ها را بعد از بازگشت از system call بازیابی میکند. و هر error code ای که از system call باز میگردد را در errorno ذخیره میکند.(در صورت رخ دادن error)
منبع: http://man7.org/linux/man-pages/man2/syscall.2.html

**گام دو:**

اضافه کردن یک سیستم کال برای هر سرویس جدیدی که نیاز داریم منطقی نیست زیرا برای هر سیستم کالی که اضافه کنیم لازم است که کرنل را مجدد make کنیم که این کار زمان بسیار زیادی طول می‌کشد از طرفی نیز تعدد سیستم‌کال های غیر استاندارد باعث می‌شود که قابلیت انتقال و portability کاهش پیدا کند چون آن سیستم کال ها روی سیستم های عادی دیگر نخواهد بود پس برنامه ما اجرا نخواهد شد.

برای اضافه کردن هر سیستم کال یک فولدر در پوشه ای که کد کرنل ما در آن قرار دارد ایجاد کردیم.سپس در داخل آن پوشه یک فایل .c ایجاد کردیم و کد مورد نظرمان را در آن نوشتیم.سپس یک make file در همین فولدر ایجاد کرده و عبارت *obj-y:=نام فایل حاوی کد .o را در آن نوشتیم پس از آن به فولدر حاوی source کرنل بازگشتیم و فایل make file را درآن باز کردیم در خط ۸۴۲ آن عبارت* ***“core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ name of folder را در آن تغییر میدهیم سپس به بخش cd arch/x86/syscalls رفته و فایل syscall_64.tbl را باز میکنیم و عبارتی به شکل زیر را به انتهای فایل اضافه میکنیم: number of last system call+1 64 name of file sys_name of file. سپس به بخش include/linux cd/ رفته و فایل syscalls.h را باز میکنیم وبه آن asmlinkage long name of function را اضافه میکنیم.سپس کرنل را کامپایل میکنیم.پس از اتمام آن یک کد تست برای فراخوانی سیستمی جدید اضافه شده می نویسیم و آن را کامپایل میکنیم. در صورت اضافه شدن صحیح سیستم کال با وارد کردن دستور dmesg میتوانیم اجرای آن را ببینیم. کد های مربوط به فراخوانی های سیستمی مورد سوال و تست آن ها به همراه فایل هایی که با تغییرات همراه بوده اند(Make file,syscalls.h,syscall_64.tbl) ضمیمه شده است.***

**گام سوم:**

سوال ۱ : سیستم کالی با نام ptrace که کوتاه شده‌ی process trace است در لینوکس وجود دارد (https://linux.die.net/man/2/ptrace ). این سیستم کال امکانی را فراهم می‌کند که یک پردازه بتواند پردازه‌ی دیگری را ردگیری کند و آن را بررسی کرده و حتی می‌تواند حافظه‌ی آن را هم تغییر دهد. این سیستم کال اغلب برای دیباگ کردن در حالتی که breakpoint قرار داده شده و به طور کلی رد گیری سیستم کال ها استفاده می‌شود. برای استفاده از ptrace باید اول به آن پردازه ای که میخواهیم ردگیری کنیم متصل شویم و این کال به صورت thread ۱ thread ۱ انجام می‌شود. تعریف کلی این سیستم کال به شکل زیر است:

```
#include <sys/ptrace.h>
long ptrace(enum __ptrace_request request, pid_t pid,
            void *addr, void *data);
```

که در آن *pid* برابر با شناسه‌ی آن thread ای است که می‌خواهیم ردگیری کنیم.مقدار بازگشتی این تابع *status* را بازمی‌گرداند که مشخص می‌کند پردازه یا ریسمان در حال رد گیری در چه وضعیتی است. *request* مشخص می‌کند که چه کاری می‌خواهیم انجام دهیم مثلا اگر مقدار آن **PTRACE_TRACEM** باشد مشخص می‌کند که پردازه باید توسط پدرش رد گیری شود در این نوع در خواست سایر پارامترهای ورودی نادیده گرفته می‌شوند. ولی مثلا اگر مقدار این درخواست برابر با **PTRACE_POKETEXT, PTRACE_POKEDATA** باشد داده‌ی *data* را در آدرس *addr* از حافظه‌ی پردازه‌ی در حال ردگیری می‌نویسد و یا دستور **PTRACE_GETREGSET** مقدار رجیسترهای پردازه‌ی در حال ردگیری را تغییر می‌دهد. که *addr* به آن رجیستر اشاره کرده و *data* اندازه و ادرس بافری است که داده ای که می‌خواهیم در رجیستر بنویسیم در آن قرار دارد. می‌تواند مقادیر بسیار متنوع دیگری هم داشته باشد.

دستور ptrace باعث می‌شود که پردازه‌ای که در حال ردگیری آن است هر بار که با یک سیستم کال به کرنل داخل شده و یا از آن خارج می شود متوقف شود تا بتوان با روی آن تغییرات لازم را داد و یا حالت کلی آن را بررسی کرد و بعد ادامه می‌دهد و این روند تکرار می‌شود.

strace برای ردگیری کردن سیستم کال‌ها و سیگنال‌ها استفاده می‌شود، در حالت ساده این دستور سیستم کال‌های استفاده شده توسط یک پردازه، نام‌های آن‌ها و سیگنال‌های بازگشتی آن‌ها به پردازه را ردگیری می‌کند. می‌توان از ptrace برای پیاده سازی آن استفاده کرد چون ما در strace می‌خواهیم که پارامترهای داده شده به سیستم کال و سیستم کال صدا شده را بیابیم همان طور که بالاتر گفتیم ptrace پس از صدا شدن سیستم کال می‌ایستد و مثلا در مدل ۸۶× پارامترها با استفاده از رجیستر ها پاس داده می‌شوند که می‌توانیم با استفاده از درخواست `PTRACE_GETREGS` از ptrace مقدار رجیسترها را کپی کنیم. و برای فهمیدن اینکه کدام سیستم کال هم صدا شده، هر سیستم کال یک عدد خاص خود را دارد که توسط رجیستر ها فرستاده می‌شود پس با خواندن مقدار آن ها نوع سیستم کال را هم متوجه می‌شویم و همان طور که گفتیم پس از پایان سیستم و بازگشت از کرنل هم ptrace متوجه شده و پردازه را متوقف می‌کند پس بازهم strace می‌تواند مقدار بازگشتی که همان سیگنال فرستاده شده به پردازه است را ردگیری کند.

سوال ۲: دلیل استفاده از کتاب خانه ها به جای دسترسی مستقیم این است که اگر هر برنامه نویس بخواهد به طور مستقیم از خود سیستم کال ها استفاده کند پس تمام برنامه نویسان باید شهود خوبی نسبت به سیستم و زیرساخت های آن و نحوه‌ی پیاده کار سیستم کال ها و سیستم کال های موجود داشته باشند که این خود کار دشواری است. از طرفی نیز چون سیستم کال ها الزاما بین تمام سیستم ها یکی نیستند و گاها وابسته به ماشین و یا سیستم عامل اند اگر بخواهیم در برنامه‌های خود به طور مستقیم

استفاده کنیم برنامه portability خود را از دست خواهد داد. پس ترجیح می‌دهیم به جای اینکه خود را نحوه‌ی پیاده سازی سطح پایین دستورات درگیر کنیم و portability برنامه خود را کم کنیم از کتابخانه‌های استاندارد استفاده کنیم تا برنامه روی تمام سیستم ها قابل اجرا باشد و نوشتن آن هم ساده تر باشد

سوال ۳:

```
void* malloc (size_t size);
void *sbrk(intptr_t increment);
void *mmap(void *addr, size_t lengthint " prot ", int " flags ,int fd, off_t offset)
```

این تابع به اندازه‌ی size بایت در حافظه فضا می‌گیرد و اشارگری به آن را برمی‌گرداند. در پیاده سازی این تابع از ۲ سیستم کال استفاده شده است. Mmap وظیفه‌ی map or unmap کردن در حافظه را دارد. سیستم کال دوم sbrk است که اندازه‌ی بخش داده‌ی برنامه‌ی ما را تغییر می‌دهد.
به طور دقیق تر به بررسی Mmap می‌پردازیم. (https://linux.die.net/man/2/mmap ) این دستور یک تناظر جدید در حافظه‌ی مجازی پردازه ایجاد می‌کند که شروع این تناظر از addr بوده و آدرس تناظر جدید را به عنوان مقدار برگشتی به ما باز می‌گرداند. prot نوع محافظت از حافظه جدید را مشخص می‌کند که آیا می‌تواند خوانده شود و یا در آن نوشته هم بشود و یا ... . flags مشخص می‌کند که آیا به روز رسانی ها در این تناظر به اطلاع سایر پردازه‌هایی که از آن استفاده می‌کنند هم برسد و یا خیر. Fd برابر با file descriptor است و تناظر ما به اندازه‌ی lengthint و با شروع از offset انجام خواهد شد.
شکل زیر برای کمک به فهم بهتر منظور از تناظر در حافظه مجازی پردازه است.

به سراغ sbrk می‌رویم. (https://linux.die.net/man/2/sbrk ) این دستور program break که به پایان بخش داده‌ی برنامه اشاره می‌کند را تغییر می‌دهد. این سیستم کال اندازه‌ی اشاره کننده به انتهای بخش داده را تا زیاد می‌کند و در صورت موفقیت مقدار قبلی program break که در واقع شروع حافظه‌ی گرفته شده‌ی جدید است را بازمی‌گرداند و در صورت عدم موفقیت مقدار ۱- برمی‌گرداند. استفاده ی مستقیم از این دستور به جای malloc پیشنهاد نمی‌شود چون پارامترهای آن برای سیستم‌عامل‌های مختلف متفاوت اند و این قابلیت انتقال برنامه را پایین می‌آورد.