زیبا امیدوار - سمانه راسخ - نازنین صبری

گام ۱: ابتدا باید کرنل لینوکس را دانلود کنیم.خوب است که از ابتدا به حالت root برویم و یا دستورات را با sudo انجام دهیم چون تعداد زیادی از دستورات به دسترسی روت نیاز دارند. سپس با استفاده از دستور ۱ آن را از حالت فشرده (زیپ) خارج می‌کنیم. سپس با استفاده از دستور cd به داخل پوشه‌ی ایجاد شده می‌رویم. در اولین قدم باید با دستور ۲ تنظیمات اولیه را انجام دهیم.این دستور به کاربر اجازه میدهد ویژگی های لینوکسی که قصد کامپایل آن را دارد،انتخاب کند.این دستوریک target در make file است. در صورتی که در هر مرحله از اجرای دستورات به یک ارور برسیم کافی است با یک سرچ ساده کتابخانه‌ای که گفته شده نصب نمی‌باشد را نصب کنیم به عنوان مثال پس از وارد کردن دستور ۲ ممکن است یک خطا رخ دهد که با وارد کردن دستور ۳ رفع خواهد شد. پس از پایان موفقیت‌آمیز دستور ۲ صفحه‌ای برای ما باز خواهد شد تا به ما اجازه دهد که تنظیمات را به صورت دستی انجام دهیم در صورتی که به این کار علاقه‌مند نیستیم کافی است این صفحه خارج شویم و اگر در پوشه‌ی boot نگاه کنیم می‌بینیم که یک سری configfile موجود است، با اجرای دستور شماره ۴ می‌توانیم یکی از این فایل‌ها را در .config کپی کرده و با دستور ۵ آن فایل به عنوان تنظیمات اولیه اعمال شود. زمانی که دستور ۵ را وارد می‌کنیم از ما می‌خواهد که یک سری تغییرات و تنظیمات را مشخص کنیم اگر دکمه‌ی enter را نگاه داریم تمام تنظیمات به حالت default اعمال خواهد شد. حال زمان کامپایل کردن کرنل است، کافی است دستور ۶ را به یکی از ۲ حالت نشان داده شده وارد کنیم، اگر از j- استفاده کنیم روی تردهای مختلف کامپایل می‌کند تا سرعت کامپایل بالا رود. پس از آن باید ماژول‌های کرنل را نصب کنیم و بعد کرنل را نصب کنیم، ایمیج رم دیسک را نداریم پس آن را می‌سازیم (دستور ۱۰) و پر می‌کنیم، سپس دستور ۹ را اجرا می‌کنیم که در پوشه‌ی بوت خواهد گشت و هر چیز جدیدی که پیدا می‌کند را به گراب اضافه می‌کند و در نهایت با اجرای دستور ۱۱ کرنل ما روی qemu بالا خواهد آمد. در دستور ۱۱پس ازsystem -مدل معماری سیستم و پس از kernel- منبع کرنل را مشخص می‌کند، initrd- دیسک ایمیج را مشخص می‌کند و m- میزان حافظه‌ای که به آن اختصاص داده‌ایم. میتوانیم قبل از دستور ۱۱دستور rebootرا اجرا کنیم ولینوکس جدید را بالا بیاوریم.

(1) tar xjf linux-.....tar.bz2
(2) make menuconfig
(3) apt-get install libncurses2-dev
(4) cp /boot/config-.... .config
(5) make oldconfig
(6) make make -j8
(7) make modules_install
(8) make install
(9) update-grub
(10) cd arch/x86/boot/
mkinitramfs -oinitrd.img-<kernel verion we are making it for>
(11) qemu-system-x86_64 -kernel bzImage -initrd initrd.img-3.6.2 -m 1GB

اگر بخواهیم کرنل را در حالت stopped بالا بیاوریم تا بتوانیم عمل debug را انجام دهیم دستور ۱۱ را به صورت زیر اجرا می‌کنیم S- سیستم را در حالت stoppedبالا می‌آورد و s- باعث می‌شود که بتوانیم با اشکال‌زدا به کرنل وصل شویم، به طور پیش‌فرض پورت ۱۲۳۴ برای این کار اختصاص داده می‌شود :

(12) qemu-system-x86_64 -s -S -kernel bzImage -initrd initrd.img-3.6.2 -m 1GB

پس از اینکه صفحه‌ی qemu باز شد با یک ترمینال دیگر به پوشه‌ی linux رفتیم و دستور gdb را وارد می‌کنیم، این دستور gdb را برای ما باز می‌کند، سپس با دستور ۱۳ به qemu وصل می‌شویم، بعد از این باید مرجع کرنل خود و symbol-file مربوطه که debugging information را در خود نگه‌داری می‌کند اضافه کنیم، symbol table در این فایل قرار دارد، که جدولی است که به دیباگر می‌گوید که هر تابع یا متغیر یا اسم یا ... در حافظه با چه بخشی در کد اصلی مرتبط است. دستورات ۱۴ و ۱۵ این کار را انجام می‌دهند. در این وضعیت زمانی که دستور ۱۶ را وارد می‌کنیم ممکن است جلو نرود و آدرس تغییر نکند، علت این است که معماری پیش‌فرض gdb با معماری سیستم ما هم‌خوانی ندارد به همین دلیل دستور ۱۷ را اجرا کرده و از میان معماری‌های موجود i386 انتخاب کردیم. حال دستور ۱۶ درست کار می‌کند. با استفاده از دستور ۱۸ بریک‌پوینت گذاشتیم.پس از انجام دستورات زیر با وارد کردن دستورات زیر با پیغام no source availableمواجه شدیم.به همین علت دستورات را به شکل دستور ۱۹تا۲۳را وارد کردیم.سپس با پیغام packet replay is too long مواجه کردیم.برای رفع این مشکل دستورات ۲۴تا ۲۷ را وارد کردیم.خطی که در آن break pointگذاشته بودیم به ما نشان داده شد و با وارد کردن n میتوانیم به خط های بعدی برویم.

(13) target remote localhost:1234 (14) source ./kernel (15) symbol-file vmlinux (16) ni
(17) set architecture i386 (18) b <function_name> - break +<offset> (19) set arch i386:x86-64:intel
(20) target remote localhost:1234 (21) symbol-file vmlinux (22) break <function_name) (23) continue
(24) disconnect (25) set arch i386:x86-64 (26) target remote localhost:1234 (27) la src

گام ۲ - یک تابع: در ابتدا به بررسی تابع memparse در kernel v3.14.79 پرداختیم. وظیفه‌ی کلی این تابع تبدیل یک string به یک عدد بدون علامت (long long) است. نحوه‌ی کار آن این است که با صدا کردن تابع simple_strtoull قسمت عددی را بدست می‌آورد سپس به اولین حرف پس از عدد نگاه می‌کند و با توجه به اینکه گیگ، مگ یا کیلو است به تعداد لازم عدد را شیفت می‌دهد. در آخر اگر پوینتری برای رشته‌ی بعدی داده شده باشد (تعریف شده باشد، null نباشد) انتهای این رشته را به عنوان ابتدای بعدی در نظر می‌گیرد و مقدار تبدیل شده را برمی‌گرداند.

Memparse: return value: unsigned long long عدد تبدیل شده parameters: پوینتری به شروع این رشته و اشارگری به پوینتر شروع رشته‌ی بعدی

همان طور که گفتیم تابع simple_strtoull در این تابع صدا می‌شد. به طور خلاصه این تابع مبنای رشته‌ی داده شده را با استفاده از تابع simple_guess_base پیدا می‌کرد و بعد با توجه به مبنا رشته را به عدد تبدیل می‌کند. در صورتی که مبنا ۱۶ باشد اشارگر به سر رشته را ۲ تا جلو می‌برد. (تا از 0x اول بگذرد). سپس با استفاده از تابع isxdigit برای هر کاراکتر در رشته شرط رقم بودن در آن مبنا را بررسی می‌کند، اگر شرط برقرار بود و رقم از مبنا بزرگتر نبود، آن را به معادل عددی‌اش در مبنای ۱۰ تبدیل می‌کند و تا انتهای رشته ادامه می‌دهد. اگر isxdigit برقرار نبود ادامه‌ی رشته را به عنوان بخشی از عدد در نظر نمی‌گیرد.

Simple_strtoull: return value: unsigned long long: عدد تبدیل شده بدون در نظر گرفتن بزرگی (M, G, K) parameters: cp= اشارگری به شروع

endp = اشارگری به پوینتر به اولین حرف پس از عدد که البته در تابع مقدار دهی می‌شود و زمانی که به عنوان پارامتر داده شده، صرفا تعریف شده است رشته

base= مبنای رشته‌ای است که می‌خواهیم تبدیل کنیم که هر بار ۰ داده می‌شود تا خود تابع مقدار آن را محاسبه کند

همان طور که گفتیم این تابع، تابع simple_guess_base را صدا می‌کرد. که این تابع مبنا را حدس می‌زند. به این صورت که اگر کاراکتر اول رشته ۰ باشد و کاراکتر دوم × باشد و کاراکتر بعدی از ارقام مجاز مبنای ۱۶ باشد مبنا را ۱۶ اعلام می‌کند، در غیر این صورت اگر فقط کاراکتر اول ۰ باشد ولی حتی یکی از ۲ شرط دیگر برقرار نباشد مبنا را ۸ اعلام می‌کند و اگر کلا کاراکتر اول ۰ نباشد مبنا را ۱۰ اعلام می‌کند. این تابع برای انجام این کارها از تابع isxdigit کمک می‌گیرد.

Simple_guess_base: return value: unsigned int مبنا parameters: cp = اشارگر به ابتدای رشته

تابع isxdigit که بالاتر از آن استفاده شد مقدار عددی یک کاراکتر را به عنوان ورودی می‌گیرد و بررسی می‌کند که آیا این مقدار یکی از اعداد ۰ تا ۹ یا حروف a تا f (یعنی ارقام مجاز مبنای ۱۶) هست یا خیر، در صورتی که باشد عدد ۱ و در غیر این صورت ۰ را برمی‌گرداند یعنی مقدار بازگشتی آن حالت true false دارد.

گام سوم -مرحله ۱- سوال ۰: کلمه‌ی بوت در کامپیوتر به معنی load شدن سیستم‌عامل در حافظه‌ی اصلی و یا رَم کامپیوتر است. پس از اینکه سیستم‌عامل load شد آماده‌ی اجرای برنامه‌ها توسط کاربران است. (منبع: http://searchwindowsserver.techtarget.com/definition/boot) بوت شدن با توجه به منابع دارای ۵ مرحله (یا ۶) است: ۱- BIOS: این کلمه خلاصه‌ی basic input/output system است. پس از فشردن دکمه روشن شدن cpu در ROM به دنبال دستوری که باید اجرا کند می‌گردد، ROM شامل دستور پرش است که cpu را به بخش BIOS می‌برد. در BIOS لیست تمام ابزارهای ورودی و خروجی موجود برای سیستم وجود دارد. و در واقع BIOS مراحل شروع کار سخت‌افزارهای خاص سیستم را انجام می‌دهد، زمانی که سخت‌افزارهای واجب برای بوت شدن initialize شدند به مرحله‌ی بعدی می‌رویم.به طور دقیق‌تر زمانی که بوت لودر را از دیسک یا هر جایی که در آن هست در حافظه لود کردیم کنترل را به آن می‌سپاریم. ۲- Boot loader: Boot loader که به اسم boot system نیز شناخته می‌شود در واقع برنامه‌ی کوچکی است که سیستم‌عامل را در حافظه قرار می‌دهد و کنترل را به آن می‌سپارد. این برنامه اغلب منوای از گزینه‌های بوت موجود را به کاربر نشان می‌دهد، در صورت انتخاب نشدن چیزی توسط کاربر یا نبودن بیش از یک گزینه، Boot loader کرنل را در حافظه load می‌کند. خود Boot loader شامل ۲ بخش است. + MBR = master boot record: این sector اول هارد دیسک است که ۵۱۲ بایت از آن را اشغال می‌کند. که حدود ۴۳۰ بایت اول آن Boot loader اولیه است. ۶۴ بایت بعدی آن به partition table تعلق دارد. یک partition table یک ساختار داده است که اطلاعاتی را برای سیستم‌عامل درباره‌ی تقسیم‌بندی (بخش‌بندی) هارد دیسک به بخش‌های اصلی‌اش می‌دهد. ۶ بایت بعدی نیز به MBR validation timestamp متعلق است. MBR در واقع GRUB که بخش دیگر Boot loader است را load و اجرا می‌کند. (از آنجایی MBR به مفاهیم و اطلاعات فایل سیستم‌ها آگاه نیست نمی‌تواند به طور مستقیم کرنل را لود کند. برای اطلاع درباره‌ی فایل سیستم‌ها در لینوکس به لینک http://www.tldp.org/ldp/intro-linux/html/sect_03_01.html مراجعه کردیم.). + GRUB = Grand unified boot loader: بخش ۱.۵ گراب در ۳۰ کیلوبایت پس از MBR و قبل از بخش‌بندی ۱ هارد دیسک قرار دارد که اطلاعات مربوط به deriver و module های فایل سیستم‌ها را در خود نگه‌می‌دارد. بخش ۲ از گراب وظیفه‌ی load کردن کرنل را دارد. در این بخش است که لیست کرنل‌ها در صورت وجود برای کاربر نمایش داده می‌شود. پس از این مرحله کنترل به کرنل داده می‌شود. ۳- Kernel: کرنل درصورتی که به صورت فشرده شده باشد خود را از این حالت خارج کرده و سریعا configurationهای مربوط به سخت‌افزار و اختصاص دادن حافظه به سیستم را انجام می‌دهد، درایورهای موجود و لازم را لود می‌کند، به دنبال هارد دیسک‌های موجود می‌گردد، تمام فضای اختصاص داده شده به disk image را پاک می‌کند، بخش root خود را به عنوان فقط خواندنی مشخص می‌کند و processای به نام init را اجرا می‌کند. ۴- Init process: این برنامه اولین برنامه‌ی بخش user-space است که اجرا می‌شود و اولین برنامه‌ای است که با لایبری‌های C کامپایل می‌شود. این فایل نیازهای سیستمی را مشخص می‌کند. ۵- runlevel scripts: یک runlevel به معنی وضعیتی است که می‌خواهیم سیستم‌عامل در آن بالا بیاید، به عنوان مثال برای یک سیستم لینوکس این حالت می‌تواند، تک کاربر، چند کاربری، با command line interface، با GUI و یا ... باشد. با توجه به runlevel مربوطه فایل init متناسب با آن runlevel scripts مربوطه صدا می‌کند. * حال اگر همه چیز تا اینجا خوب پیش رفته باشد باید صفحه‌ی ورود را ببینیم.

(منابع: https://en.wikipedia.org/wiki/Linux_startup_process - http://www.linfo.org/runlevel_def.html - http://www.golinuxhub.com/2014/03/step-by-step-linux-boot-process.html - http://www.linfo.org/partition_table.html - http://linuxlecture.com/boot-process-in-linux/ - http://searchdatacenter.techtarget.com/definition/boot-loader-boot-manager)

گام سوم - مرحله۱- سوال ۱: علت اینکه از زبان اسمبلی استفاده می‌کنیم این است که برخی کارها فقط در زبان اسمبلی امکان پذیر اند به عنوان مثال با زبان اسمبلی می‌توان به رجیسترهای وابسته به ماشین و ورودی و خروجی دسترسی پیدا کرد، در حالی که این کار در زبان C امکان پذیر نیست و معادلی ندارد. و از این‌جایی که دستورات این فایل وابسته به ماشین و cpuای که روی آن اجرا شده است باید به زبان اسمبلی نوشته‌شود. اگر بخواهیم این کدها را مثلا به زبان c بنویسیم باید کتابخانه هایی که کارهای مربوط به بوت شدن را انجام می دهند هم بنویسیم. از آنجایی که این کتابخانه ها غیر از کارهای بوت شدن لینوکس استفاده ی دیگری ندارند،بهتر است آن ها را به زبان ماشین بنویسیم. علاوه بر آن چون می‌توانیم کدی که به صورت کامل برای آن سخت‌افزار خاص ساده سازی شده است و متناسب شده است بنویسیم، سرعت اجرای کد از هر زبان دیگری بیشتر است. یک قابلیت دیگر این است که می‌توانید به مدهای برنامه نویسی خاص دسترسی پیدا کنید. علاوه بر آن امکان دارد که پس از روشن شدن سیستم، سیستم آماده‌ی اجرای برنامه به زبان C نباشد، علت این آماده نبودن می‌تواند این باشد که پروسسور در حالت ۶۴ یا ۳۲ بیت نباشد یا اینکه استکی یا کتابخانه‌هایی که برای کد C نیاز است آماده نباشد. در این فایل مقداردهی‌های اولیه‌های مورد نیاز برای بوت شدن انجام می‌شود و در انتهای فایل اگر مشکلی

پیش نیامده باشد تابع main صدا زده می‌شود و وارد کدهای C می‌شویم. علت قرار داشتن این فایل در اینجا این است که فایل اولیه‌ی شروع کار کرنل برای معماری خاص x86 است و برای هر معماری متناسب با نوعش فایلی مشابه این فایل در پوشه‌ی مربوط به خودش وجود دارد. یعنی به ازای هر معماری یک زیر پوشه وجود دارد.

گام سوم - مرحله ۱ - سوال ۲: به سراغ تابع main در /arch/x86/boot/main.c رفتیم، به طور خلاصه این تابع وظیفه‌ی بوت کردن سیستم و اداره کردن آن و مقداردهی های اولیه را به عهده دارد و مثل یک واسط بین real mode و protected mode عمل کند. از کپی کردن هدرهای بوت شروع می‌کند و تا رسیدن و داخل شدن به protected mode می‌رود و در این میان کارهایی مثل چک کردن سخت‌افزارهای موجود تا انتخاب مد اجرای پردازنده و ... را انجام می‌دهد. اگر بخواهیم به صورت گام به گام توضیح دهیم: ۱- بوت هدرها را در صفحه‌ی ۰ حافظه قرار می‌دهد. ۲- کنسول ابتدایی زمان بوت شدن سیستم را ایجاد می‌کند ۳- اطمینان حاصل می‌کند که پشتیبانی سخت‌افزاری لازم را داریم و cpu پشتیبانی لازم برای بالا آمدن کرنل ما را ایجاد می‌کند. ۴- به بایوس حالتی که می‌خواهیم cpu در آن اجرا شود را اطلاع می‌دهد ۵- از وجود واحد حافظه اطمینان حاصل می‌کند ۶- نرخ تکرار کیبرد را در حالت ماکزیمم قرار می‌دهد ولی خودش هم علت این کار را نمی‌داند ۷ -یک سری پرس و جو از سیستم‌های اطراف خود می‌کند که وضعیت هر یک را پیدا کند ۸- مد ویدیو را مشخص می‌کند که می‌تواند چیزی مثل graphic mode یا text mode باشد ۹- به حالت پروتکتت می‌رود. این تابع مقداری را باز نمی‌گرداند.

گام سوم - مرحله ۱ - سوال۳: خیر برای معماری‌های مختلف متفاوت نیست، مثلا تابع‌های زیر برای معماری‌های مختلف اند ولی تمام آن‌ها داخل خود این تابع را صدا می‌کنند. مقدار بازگشتی آن void است ولی در برخی از جاها به unsigned long تبدیل شده بود! دسترسی کرنل مود است. این تابع بوت را باز کرده کرنل را از ادرس گفته شده‌ی آن آورده و باز کرده و لود می‌کند و اگر مشکلی نبود با صدا کردن تابع ران کرنل آن را ران می‌کند.

__INIT START_KERNEL - X86_64_START_RESERVATIONS - I386_START_KERNEL - START_UML -> START_KERNEL_PROC - DEFINE_PER_CPU, BOOT_PC - GDBSTUB - METAG_START_KERNEL - DEBUG_STUB_INIT

گام سوم - مرحله ۱ - سوال۴:یک کنسول ساده و ابتدایی راه می‌اندازد که موقتا پیام‌های مربوط به بوت سیستم را در آن نمایش می‌دهد.

گام سوم - مرحله ی دوم: واسط(interface) بین برنامه های سطح کاربر وسیستم عامل با مجموعه ای از دستورالعمل های توسعه یافته که سیستم عامل آن ها را در اختیار میگذارد تعریف شده است.این دستور العمل های توسعه یافته با عنوان فراخوان سیستمی شناخته شده اند. هرگاه یک نرم‌افزار سطح کاربر نیاز به دسترسی به منابع سیستم و سخت افزار را داشته باشد، یکی از توابع درون سیستم عامل را فراخوانی می‌کند. که به این عمل فراخوان سیستم میگویند. .در واقع system call ها لایه ای بین سخت افزار و فضای کاربر هستند.فراخوانی سیستمی معمولا از طریق فراخوانی های تابع که در کتابخانه ی c تعریف شده است؛قابل دسترسی است.اگر یک فرایند مشغول اجرای یک برنامه در user mode باشد ونیاز به یک سرویس سیستمی مانند خواندن داده از فایل داشته باشد مجبور خواهد بود یک تله-trap- یا فراخوان سیستمی را اجرا کند تا کار را به سیستم عامل بسپارد.سپس سیستم عامل با توجه به پارامترها متوجه درخواست فرایند صدازده میشود.آنگاه فراخوان سیستمی را اجرا میکند.سپس روال مربوطه در هسته انجام میشود.

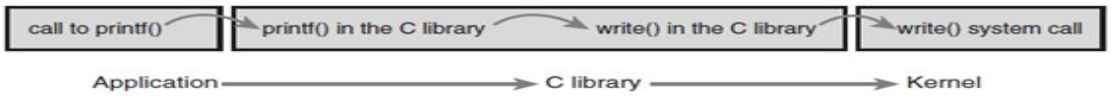


Figure 5.1 The relationship between applications, the C library, and the kernel with a call to `printf()`.

System call ها معمولا به آرگومان ها نیاز دارند.مثلا open() نیاز دارد که به کرنل بگوید دقیقا کدام فایل را بخواند.

System call ها یک داده‌ی بازگشتی از نوع long باز میگرداند که موفقیت یا خطا را مشخص میکند.کتابخانه ی c هنگامی که system call یک error بازمیگرداند؛یک error codeخاص در متغیر global –errno می نویسد. در حالت ارور اکثر سیستم کال ها یک مقدار منفی باز میگردانند که این مقدار توسط wrapper ها مخفی میشود و عدد ۱- به سیستم اصلی باز گردانده میشود.خروجی سیستم کال ها در صورت موفقیت بستگی به نوع سیستم کال دارد اما اغلب صفر است.همه ی system callها یک system call numberدارندکه توسط kernelوuserspaceشناخته شده است.برای مثال هر دو میدانند که که system call با شماره ی ۱۰مربوط به open()است یا مثلا system call با شماره ی ۱۱ مربوط به read است.system call number ها در رجیستر نگه داری میشوند. kernel یک لیست از همه ی این رجیسترها را در system call table نگه داری میکند تا بتواند از آن ها استفاده کند.برای فضای کاربر امکان پذیر نیست که kernel codeرا به طور مستقیم اجرا کنند.برنامه های فضای کاربر باید به کرنل سیگنال ارسال کنند که میخواهند یک system callاجرا کنند و سپس به حالت کرنل سوییچ کنند.مکانیسم فرستادن سیگنال به کرنل یک وقفه ی نرم افزاری است.یک exception ایجاد میشود.سیستم به کرنل مد تغییر میکند و exception handler را اجرا میکند.در این مورد exception handler در واقع همان system call handler است.وقفه ی نرم افزاری تعریف شده در intrupt number 128،x86 است که از طریق دستور $0x80اجرا میشود.این مسئله باعث سوییچ به کرنل و اجرای exception vector 128 که در واقع همان system call handler میشود.

وارد شدن به فضای کرنل به تنهایی کافی نیست به این دلیل که فراخوانی های سیستمی چندگانه وجود دارند که همه ی آن ها به روش مشابه وارد کرنل میشوند .بنابراین برای تشخیص نوع فراخوان سیستمی باید system call number به کرنل فرستاده شود.این مقدار در رجیستر eax نگه داری میشود.system call handler. مقدار eax را میخواند.تابع system-call() اعتبار system call number را با مقایسه ی آن با NR_syscalls میسنجد.اگر مقدار آن بزرگتر یا مساوی NR-syscalls باشد تابع –ENOsys را بازمیگرداند در غیر اینصورت سیستم کال مشخص شده ی زیر استفاده میشود:

(Call *sys_call_table(%rax,8

هر system call یک رفتار مشخص دارد.مثلا ساده ترین فراخوان سیستمی getpidکه id فرایند فعلی را باز میگرداند به صورت زیر است.این فراخوانی هیچ آرگومانی دریافت نمیکند.

(SYSCALL_DEFINE0(getpid

```
}
return task_tgid_vnr(current); // returns current->tgid
}
```

علاوه بر system call numberاغلب syscall ها به یک یا پارامترهای بیشتری نیاز دارند که باید به آن ها فرستاده شود.در واقع فضای کاربر باید پارامترها را برای کرنل بفرستد.در x86،این پارامترها در رجیسترهای ebx-ecx-edx-esi-edi ذخیره میشوند.همچنین مقدار بازگشتی که به فضای کاربر فرستاده میشود در رجیستر eax ذخیره میشود.

منابع:کتاب LOVE کتاب سیستم عامل پارسه https://0xax.gitbooks.io/linux-insides/content/SysCall/syscall-1.html

گام سوم-مرحله 3:

سوال۱:File descriptor: فایل ها باید قبل از خواندن یا نوشتن باز شوند.در همان زمان،اجازه دسترسی بررسی میشود.اگر اجازه داده شده باشد،سیستم یک عدد صحیح کوچک به نام file descriptor برمیگرداند که برای عملیات بعدی مورد استفاده قرار میگیرد،اما اگر دسترسی منع شده باشد، یک کد خطا (۱-) برمیگرداند.
این file descriptor را میتوان همانند یک اشاره گر در غالب یک آرگومان به دیگر توابع داد.
کلیه process ها در لینوکس که با fileها سر و کار دارند برای شناسایی fileیی که روی آن کار میکنند از اشاره گر یا توصیف کننده file استفاده میکنند.در این روش کلیه فراخوانی های سیستم که با file تعامل دارند یک file descriptor می گیرند یا برمیگردانند.file descriptor معمولا بین ۰ تا ۲۵۵ است.
این عدد شاخص است که محل file را در جدول file descriptor توصیف میکند که این جدول برای هر process جداگانه تعریف میشود.
هر برنامه در حال اجرا با ۳ file باز شده شروع به کار میکند.

Description	File Number	Short Name	Descriptive Name
Input from the keyboard	0	stdin	Standard In
Output to the console	1	stdout	Standard Out
Error output to the console	2	Stderr	Standard Error

سوال۲:تابع ()pipe یک pipe ایجاد می کند که از ۲ file descriptor تشکیل شده است که در ارتباط با دو سر pipe می باشند(read end,write end).این در واقع مثل file نیست.kernel داده را از write end می خواند و آن را buffer می کند و آن را به read end منتقل می کند.این مشخص می کند که چرا ()pipe دو file descriptor ایجاد می کند.writer همه ی داده هایی را که نیاز است در write fd مینویسد و fd را میبندد و همچنین trigger می کند که EOF فرستاده شود.
معمولا reader به خواندن داده ها ادامه می دهد تا با EOF مواجه می شود و سر آن را می بندد.در این روال بازه زمانی وجود دارد که write fd بسته است ولی هنوز داده در buffer،pipe مانده است تا reader آن را بخواند.

سوال۳:توابع رایجی که از file descriptor استفاده میکنند عبارتند از open,close,read,write.در اینجا به عنوان مثال تابع close را بررسی می کنیم.close برای جدا کردن استفاده یک file descriptor از یک process استفاده می شود.در واقع هدف delete کردن file descriptor می باشد.وقتی یک process پایان می یابد هر open file descriptor توسط kernel به صورت اتوماتیک بسته می شود. ;(int close(int d در این تابع d، file descriptor می باشد و خروجی تابع یک عدد int است که در صورت 0 بودن نشانگر انجام شدن عملیات با موفقیت می باشد و اگر 1- باشد نشانگر بروز خطا می باشد.

منبع:https://www.classes.cs.uchicago.edu/archive/2017/winter/51081-1/LabFAQ/lab2/fileio.html

سوال۴: تابع open پس از باز کردن یک file، یک file descriptor که یک عدد صحیح یکتا مثبت میباشد برمی گرداند که در واقع indexیی میباشد در آرایه ای از فایل های باز شده که توسط kernel نگهداری می شود.اتفاقی که وقتی یک فایل باز می شود می افتد این است که kernel از اطلاعات path استفاده می کند تا file descriptor را با چیزی که API مناسب read و write را تامین می کند map کند. وقتی دستور open برای یک device است عددهای major و minor گره device باز شده اطلاعات مورد نیاز kernel را برای یافتن device driver درست و کامل کردن mapping به آن میدهد .kernel بعد از آن می داند که چگونه call های بعدی مانند read را به تابع های زیرین که توسط device driver تامین شده است route کند.برای عملیات های non-device file هم همین گونه است با این که لایه های بیشتری در این بین هست.abstraction اینجا mount point میباشد.mount کردن file system دو هدف از mapping دارد یکی این که file system ، آن underlying device را که حافظه تامین می کند بشناسد و نیز kernel میداند که فایل باز شده تحت mount-point باید به file system driver هدایت شود.file systemها هم مانند device drivers به API عمومی file system که توسط kernel تامین میشود نوشته میشوند.

منبع:https://www.bottomupcs.com/file_descriptors.xhtml

گام چهارم:

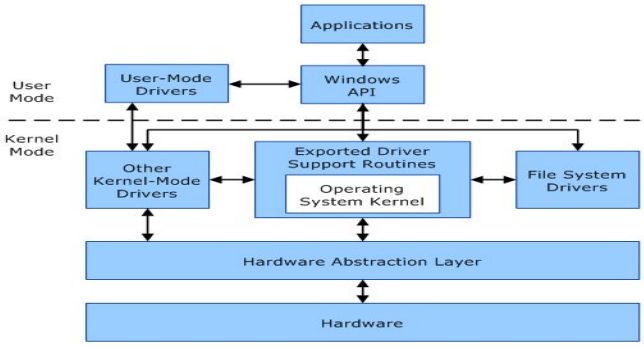
Kernel mode: در حالت کرنل، کد در حال اجرا دسترسی کامل و بدون محدودیت به سخت افزار پایه دارد. این حالت میتواند هر دستورcpu را اجرا کند و هر آدرس حافظه ای را ارجاع دهد.کرنل مود به طور کلی برای توابع سطح پایین و با قابلیت اعتماد بالا سیستم عامل مورد استفاده می شود.crashدر کرنل مود فاجعه آفرین است و میتواند pcرا متوقف کند.

User mode: در حالت یوزر مود،کد در حال اجرا توانایی دسترسی مستقیم به سخت افزار یا ارجاع حافظه ندارد.کد درحال اجرا باید دسترسی به سخت افزار و یا حافظه را به APIهای سیستم محول کند.با توجه به حفاظت ها و پروتکت هایی که در این حالت انجام میشود،crash در یوزر مود همیشه قابل بازیابی و حل شدن است.

(منبع: https://blog.codinghorror.com/understanding-user-and-kernel-mode/)

بسته به کد در حال اجرا پردازنده بین این دو حالت سوییچ میکند. به طور کلی برنامه ها در یوزر مود و اجزای اصلی سیستم عامل در کرنل مود اجرا میشوند.در حالی که بیشتر درایور ها در حالت کرنل اجرا میشوند اما بعضی از درایور ها نیز ممکن است در حالت یوزر اجرا شوند.زمانی که از یک برنامه ی یوزر مود استفاده میکنیم؛سیستم عامل یک فرایندی برای آن برنامه ایجاد میکند.این فرایند برای برنامه یک فضای آدرس مجازی خصوصی ویک جدول مدیریت خصوصی فراهم میکند.به علت این خصوصی بودن فضای آدرس مجازی برنامه یک برنامه نمیتواند داده ای را که مربوط به برنامه ی دیگر است تغییر دهد.هر برنامه به صورت مجزا انجام میشود و این مسئله باعث میشود اگر در یک برنامه crash اتفاق افتاد برنامه های دیگر تحت تاثیر آن قرار نگیرند.

تمام کدهایی که در کرنل مود اجرا می شوند؛یک فضای آدرس مجازی واحد را به اشتراک میگذارند.این مسئله به این معنی است که driver کرنل مود از سایر driver ها و سیستم عامل جدا نیست.اگر یک driver کرنل مود به صورت تصادفی در آدرس مجازی اشتباه بنویسد؛داده ای که متعلق به سیستم عامل و یا سایر driver ها است ممکن است به خطر بیفتد.اگر یک driver در حالت کرنل مود دچار crash شود؛کل سیستم عامل دچار crash میشود.انتقال میان این دو حالت هزینه بسیار بالایی دارد،به همین خاطر است که برنامه هایی که exception میدهند بسیار کند هستند چون بسیار سخت است که از سطح خطا به سطح دیگر بیاید.تصویر زیر ارتباط بین یورز مود و اجزای کرنل مود را نشان میدهد:



https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode

Real mode: نسخه ی اورجینال IBM PC فقط می توانسته 1MB از حافظه ی سیستم را آدرس دهی کند و نسخه های اصلی DOS که برای کار روی این طراحی شده بودند با این فرض طراحی شده بودند. DOS به طور ذاتی یک سیستم عامل single-tasking می باشد یعنی فقط اجرای یک برنامه را در زمان میتواند رسیدگی کند.اگر سیستم در حالت real باشد برنامه ها می توانند مستقیما به تمام خانه های آدرس پذیز حافظه و آدرس های I/O و وسایل متصل ارتباط برقرار کنند. همه ی پردازنده ها حالت real را دارند ولی در واقع DOS و برنامه های استاندارد DOS از این حالت استفاده میکردند.

Protected mode: این حالت بسیار قوی تر از حالت real میباشد و در تمام سیستم عامل های multitasking استفاده میشود.در این حالت در صورتی که برنامه با خروجی ها کار داشته باشد داده ها ارسال میشود ولی سیستم عامل داده ها را جمع می کند و آنها را به منظور ایجاد عدالت بین برنامه ها مدیریت میکند.در این حالت دسترسی کامل به حافظه ی سیستم موجود میباشد.توانایی مدیریت اجرای هم زمان چند برنامه وجود دارد(multitasking). پشتیبانی از Virtual memory که به سیستم این اجازه را میدهد تا درصورت نیاز به حافظه بیشتر از هارد دیسک استفاده کند همچنین دسترسی سریع تر به حافظه و درایور سریعتر برای انتقال I/O را نسبت به حالت real دارا میباشد.به این دلیل به این حالت protected گفته می شود که هر کاری که بخواهد اجرا بشود حافظه ی assign شده به خود را دارد و برنامه ها به حافظه ی دیگر برنامه ها دسترسی ندارند و از برخورد با دیگر برنامه ها جلوگیری میشود.

Device Drivers: در کامپیوتر یک device driver یک برنامه کامپیوتری است که روی یک سخت‌افزار خاص که به کامپیوتر وصل شده است عملیات انجام دهد یا آن را کنترل کند، driver یک رابط کاربری نرم‌افزاری برای یک بخش سخت‌افزاری به وجود می‌آورد که به سیستم‌عامل و سایر برنامه‌ها اجازه می‌دهد که از توابع آن سخت‌افزار استفاده کند بدون آنکه نیاز باشد به جزئیات آن سخت‌افزار تسلط داشته باشد. driver با device با استفاده از busهای کامپیوتر و یا سایر کانال‌های ارتباطی که سخت‌افزار به وسیله آن‌ها به کامپیوتر وصل شده ارتباط برقرار می‌کند. زمانی که یک برنامه یک روتین را در این device صدا می‌کند driver مربوطه یک دستور را برای device ارسال می‌کند. زمانی که device اطلاعاتی/داده‌هایی را برای driver بازمی‌فرستد، ممکن است که driver یک روتین را در برنامه‌ای که روتین ابتدایی را صدا کرده بود صدا کند. Driverها به سخت‌افزار بستگی دارند و با هر سیستم‌عامل هم متفاوت اند. در واقع وظیفه‌ی کلی درایورها این است که یک abstraction از سخت‌افزار مربوطه ایجاد کند تا سایر برنامه‌ها بتوانند با استفاده از APIها با آن سخت‌افزار در ارتباط باشند. driverها در kernel modeاجرا می‌شوند.

Software Driver: برخی driverها اصلا به بخشی از سخت‌افزار وصل نیستند، این درایورها در kernel modeاجرا می‌شوند، فرض کنید نیاز داریم که نرم‌افزاری بنویسیم که به ساختارداده‌های سیستم‌عامل دسترسی داشته باشد، در این صورت برنامه را به ۲ بخش تقسیم می‌کنیم، بخش اول در user mode اجرا می‌شود و application نام دارد و رابط کاربری را تامین می‌کند، بخش دوم در kernel mode اجرا می‌شود و به بخش‌های داخلی سیستم‌عامل دسترسی دارد و software driver نام دارد.

منابع: https://stackoverflow.com/questions/313700366/what-is-the-main-difference-between-drivers-and-user-applications - https://superuser.com/questions/447048/are-drivers-part-of-the-operating-systems - https://softwareengineering.stackexchange.com/questions/349752/why-do-device-drivers-in-linux-need-to-run-in-kernel-mode - https://en.wikipedia.org/wiki/Device_driver