

فراخوانی‌های سیستمی (پایین‌رده)

آشنایی با چگونگی کارکرد فراخوان‌های سیستمی

پس از پایان این آزمایش:

- بیشتر با فراخوانی‌های سیستمی کار کرده‌اید.
- از فراخوانی‌های سیستمی در برنامه‌های خود می‌توانید بهره ببرید.
- با چگونگی کارکرد سرب‌ها و کتابخانه‌های زبان سی بیشتر آشنا خواهید شد.

پیش‌نیازها:

- آشنایی با زبان سی
- دانستن چگونگی ساخت هسته‌ی لینوکس

۱.۱ دیباچه

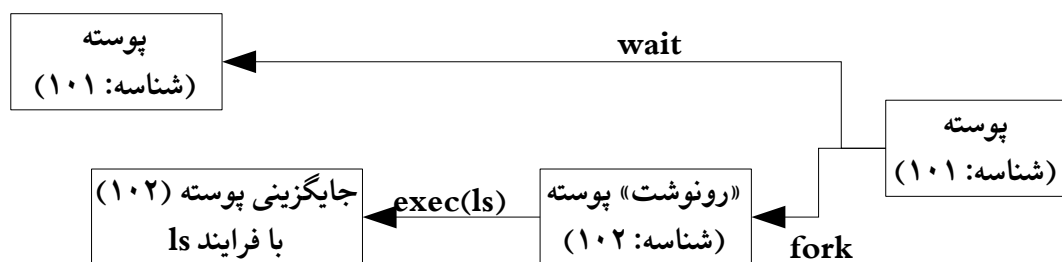
در این آزمایش، به ابزار گفتگو میان هسته و فرایندهای دیگر، «فراخوانی‌های سیستمی (پایین‌رده)»^۱، خواهیم پرداخت. با این شناخت، خواهید دانست که سیستم‌عامل به‌سادگی یک نرم‌افزار است که از «راه‌هایی ویژه» به کار با دیگر برنامه‌ها می‌پردازد، درخواست‌های آن‌ها را می‌گیرد یا آن‌ها را کارگردانی و هماهنگ می‌کند.

۱.۲ فراخوانی سیستمی چیست؟

هر برنامه‌ای با هر زبانی که نوشته شده باشد نیاز دارد که از راهی به رویه‌های پایین‌رده^۲ی سیستمی، نوشته‌شده به زبان‌های سی و اسمبلی، دسترسی پیدا کند. درخواست‌های پایین‌رده‌ی سخت‌افزاری را هر برنامه می‌تواند با «فراخوانی سیستمی» همان درخواست از «هسته‌ی» سیستم عامل بخواهد؛ نمایش هر چیز روی صفحه نمایش، حتی یک نقطه، ساخت یک پرونده، زدودن آن، باز و بسته‌کردن آن، فشردن موشواره، بسته‌شدن برنامه و ... همه نیازهای پایین‌رده‌ای هستند که درخواست برای آن‌ها باید با یک فراخوانی سیستمی به هسته داده شود.

۱.۳ فراخوانی سیستمی و کنترل فرایند (آغاز و پایان فراخوان)

فراخوان‌های سیستمی کارهایی انجام می‌دهند که فرایندها خود اجازه‌ی انجام ندارند. برای نمونه، هنگامی که دستور ls اجرا می‌شود، پوسته فراخوان سیستمی fork را صدا می‌کند تا یک رونوشت از خودش بسازد، و سپس این رونوشت پوسته خودش فراخوانی exec(ls) را اجرا می‌کند تا برنامه‌ی ls جایگزین پوسته شود. در اینجا، پدر فراخوانی waitpid را فراخوانی می‌کند و می‌ایستد تا فرزند پایان یابد.



هر برنامه (فرایند) در سیستم باید پایان می‌یابد؛ چه خودخواسته و چه ناخواسته. پس از پایان یک فرایند، سیستم‌عامل می‌بایست کنترل را به دست «راه‌انداز برنامه»^۳ یا همان «اجراکننده‌ی دستور» بازگرداند. چنانچه برنامه به‌صورت ناخواسته پایان یابد، گمان می‌شود که یک ناهنجاری در اجرای آن رخ داده است، از این‌رو، می‌بایست که

1 System Calls
2 Low Level
3 Program launcher

یک نگاشت^۴ از حافظه‌ی کوتاه‌مدت برنامه در حافظه‌ی بلندمدت نگهداشته شود تا در آینده با کمک یک برنامه‌ی «گره‌گشا»^۵ راهکاری برای آن پیدا شود.

پایان ناخواسته‌ی برنامه با ناهنجاری‌هایی با ارزش‌های ناهمسان رخ می‌دهد. از این رو، برای اینکه هسته بتواند ارزش ناهنجاری پیشامده را شناسایی کند و پاسخ درخوری به آن دهد، برنامه یک نماد هشداردهنده^۶ به برنامه‌ی فراخواننده‌ی خود برمی‌گرداند. این نماد یک شماره است و هر شماره گونه‌ی ناهنجاری پیشامده را نشان می‌دهد. برای کاهش پیچیدگی پیاده‌سازی، هسته پایان خودخواسته‌ی (هنجار) یک برنامه را با نماد و شماره‌ی «۰» می‌شناسد. سیستم عامل این نمادها را به اجراکننده‌ی دستور یا برنامه‌ی پسین می‌دهد تا پیامی به کاربر نشان داده شود و او را از شیوه‌ی پایان یافتن برنامه آگاه سازد.

گاهی یک فرایند می‌خواهد برنامه‌ی دیگری را بارگذاری و راه‌اندازی^۷ کند. چالش در این جا این است که پس از پایان برنامه‌ی بارگذاری شده، کنترل باید به کجا بازگردانده شود. زیرا زمانی که برنامه‌ی دیگری بارگذاری می‌شود، برنامه‌ی کنونی (فراخواننده) یا خودش بسته شود؛ یا بازمی‌ایستد (نگه‌داشته می‌شود)؛ یا همروند با برنامه‌ی تازه کار خود را دنبال می‌کند.

اگر می‌بایست پس از پایان برنامه‌ی تازه کنترل به برنامه‌ی کنونی بازگردد، برنامه‌ی کنونی باید نگه داشته شود (نگاشت حافظه‌ی کوتاه‌مدت آن نگهداری شود). اگر برنامه‌ی کنونی می‌بایست در کنار برنامه‌ی تازه کارش را دنبال کند، برنامه‌ی تازه را باید به شیوه‌ی چندبرنامه‌ای^۸ نوشته باشند. فراخوانی‌های fork و wait در آزمایش «فرایندها» چنین توانایی‌هایی را به ما می‌دادند.

هنگامی که چند فرایند یا کار^۹ درست می‌شوند، هسته جایگاه، ویژگی‌ها و ارزش کاری آن‌ها را شناسایی می‌کند. ویژگی‌هایی مانند اندازه‌ی حافظه‌ی کوتاه‌مدتی که فرایند نیاز دارد در همین دسته جای دارد. فراخوانی‌های سیستمی برای گذاشتن^{۱۰} و دریافت^{۱۱} این ویژگی‌ها در دسترس است. همچنین، پایان دادن به فرایندهای ناهنجار نیز با فراخوانی kill انجام می‌شود.

گاهی دو یا چند فرایند داده‌های یکسانی را به کار می‌گیرند (همرسانی^{۱۲} داده‌ها). این داده‌ها در بخش‌هایی از حافظه‌ی کوتاه‌مدت هستند برای اینکه همزمانی در دسترسی به داده‌ها پیش نیاید، باید هر فرایند بتواند در زمان کار با داده‌های هم‌رسانی شده آن‌ها را از دسترس دیگران دور نگه دارد؛ دسترسی به آن‌ها را ببندد^{۱۳}. فراخوانی‌هایی برای

-
- 4 Image
 - 5 Debugger
 - 6 Failure Code
 - 7 Execute
 - 8 Multi-Program
 - 9 job
 - 10 set
 - 11 get
 - 12 Share
 - 13 lock

بستن و بازکردن بخش‌های حافظه که جای داده‌های هم‌رسانی شده هستند در دسترس فرایندهاست.

پدیده‌ی فرایندهای هم‌زمان با ساختار سیستم‌های چندکاره^{۱۴} و تک‌کاره^{۱۵} گره خورده است. سیستم عامل داس یک سیستم تک‌کاره بود؛ در یک زمان تنها یک کار یا فرایند می‌توانست در سیستم روان باشد. در این سیستم، اجراکننده‌ی دستور فرایندی را نمی‌ساخت، تنها یک برنامه را در حافظه بارگذاری می‌کرد، بیشتر بخش‌های خودش را از حافظه بیرون می‌برد تا جا برای برای برنامه‌ی تازه باز شود، پس نشانگر (نشانه‌ی) دستور^{۱۶} را روی نخستین دستور برنامه‌ی تازه می‌گذاشت. برنامه‌ی تازه کار خود را انجام می‌داد و خودخواسته یا ناخواسته پایان می‌یافت. اگر پایان آن ناخواسته بود، یک نماد هشداردهنده بزرگ‌تر از «۰» نگه داشته می‌شد. سپس، بخشی از اجراکننده‌ی دستور که در حافظه مانده بود (یا روی آن نوشته نشده بود) کارش را دنبال می‌کند؛ نخست، بخش‌های دیگر خود را از دیسک (حافظه‌ی بلندمدت) برمی‌گرداند. پس از این، به نماد هشداردهنده رسیدگی می‌شد و آن را به کاربر یا به برنامه‌ی پسین می‌سپارد تا واکنش درخوری به آن داده شود.

لینوکس یک سیستم عامل چندکاره است. در این سیستم، پس از ورود، کاربر به یک پوسته^{۱۷} دسترسی خواهد داشت که توانایی دریافت و اجرای دستورهای کاربر را دارد. این پوسته پس از ساخت یک فرایند تازه، خودش یا بخشی از آن از حافظه بیرون نمی‌رود و می‌تواند برنامه‌ی دیگر را اجرا یا دستور دیگری دریافت کند. در اجرای برنامه‌ها، پوسته دو گونه برخورد انجام می‌دهد؛ یا تا پایان آن‌ها می‌ایستد و یا آن‌ها را «در پس زمینه» اجرا می‌کند و خودش به دریافت دستور دیگر می‌پردازد. در برخورد دوم، برنامه‌ی تازه با یک فراخوانی سیستمی fork و سپس فراخوانی exec اجرا می‌گردد. سپس، پوسته برنامه را در پس زمینه اجرا می‌کند و آماده‌ی دریافت دستور دیگری می‌شود. در این زمان دستگاه صفحه کلید تنها در دست پوسته خواهد ماند. همچنین، کاربر می‌تواند با کمک دستورهای پوسته ویژگی‌های برنامه مانند ارزش اجرایی آن را دستکاری کند. پس از اینکه فرایند کار خود را به پایان رساند، یک نماد هشدار «۰» یا بزرگ‌تر از «۰» به فراخواننده‌ی خود برمی‌گرداند. پوسته و دیگر برنامه‌ها می‌توانند این نماد را بگیرند و برپایه‌ی آن کاری انجام دهند.

۱.۴ فراخوانی‌های سیستمی سامانه‌ی پرونده^{۱۸}

کار با فایل‌ها یک روند پایین‌رده (نزدیک سخت‌افزار) است، از اینرو، از دسته‌ی کارهایی است که هسته‌ی سیستم عامل باید آن را انجام دهد. برای ساختن، زدودن و باز و بستن فایل‌ها و نیز برای خواندن از، نوشتن در آن‌ها، یا جابجایی آن‌ها فراخوانی‌های سیستمی در دسترس است. برخی از این کارها، برای پوشه‌ها هم انجام می‌شود. برای دستکاری ویژگی‌های گوناگونی از فایل‌ها مانند نام، گونه، داده‌های امنیتی نیز فراخوانی‌های در دست است.

-
- 14 Multi-tasking systems
 - 15 Single-tasking system
 - 16 Instruction pointer
 - 17 shell
 - 18 Filesystem

۱.۵ فراخوانی‌های دسترسی به دستگاه‌ها

برای جلوگیری از کارشکنی فرایندهای تراز کاربر در کار سخت‌افزارها، فرایندها نمی‌توانند آزادانه و خودسرانه با این دستگاه‌ها و نیز توانایی‌های دیگر هسته کار کنند. فراخوانی‌های سیستمی کارکرد میانجی‌گری میان هسته و فرایندها را خواهند داشت و فرایند درخواست خود را نخست به آنها می‌دهد. به زبانی ساده، فراخوانی‌های سیستمی ابزاری میان فرایندهای تراز کاربر و هسته هستند و مانند یک پیغام‌رسان میان این دو کار می‌کنند و توانایی‌های زیر را به برنامه‌های کاربردی می‌دهند:

- دسترسی مهارشده به سخت‌افزارها؛
- ساخت فرایند فرزند و گفتگو با فرایندهای دیگر؛
- توانایی درخواست دیگر دستگاه‌های سیستم عامل.

فرایندها درخواست می‌دهند و هسته یا درخواست آنها را می‌پذیرد و برمی‌آورد یا با برگرداندن یک پیغام هشدار آن را نمی‌پذیرد.

هر فرایند برای انجام کار خود به چند دستگاه نیاز دارد. این دستگاه‌ها یا سخت‌افزاری هستند (مانند گرداننده‌های سخت‌افزاری^{۱۹}) یا نرم‌افزاری هستند (مانند پرونده‌ها). سیستم عامل چگونگی دسترسی به این دستگاه‌ها را سرپرستی می‌کند. در سیستمی که چند کاربر^{۲۰} دارد، نخست، هر کاربر باید درخواستی برای بهره‌مندی از هر دستگاه را بدهد و پس از کار با دستگاه باید آن را رها سازد. این دو کار (دردست‌گرفتن دستگاه و رهاسازی آن)، مانند باز و بسته‌کردن فایل‌ها هستند. پس از دردست‌گرفتن دستگاه، فرایند از آن می‌خواند یا بر آن می‌نویسد، درست مانند کار با فایل‌ها. همسانی میان کار با دستگاه‌های ورودی-خروجی (IO) و کار با فایل‌ها چنان زیاد است که برخی بُن‌سامان‌ها، مانند یونیکس و لینوکس، هر دو را با یک دیدگاه فایل-دستگاه نگاه می‌کنند و فراخوانی‌های سیستمی یکسانی برای هر دو دارند. گاهی دستگاه‌های ورودی-خروجی با نام‌های فایلی، نشانی پوشه‌ای یا ویژگی‌های فایلی در دسترس هستند. برای نمونه، در لینوکس در زیرشاخه‌ی `dev` از ریشه‌ی ساختار درختی فایلی (`/dev`) می‌توان به پارتیشن‌های (تکه‌های) دیسک‌ها، حافظه‌های قابل حمل و گرداننده‌ی سی‌دی دسترسی داشت.

با چنین سازوکاری، یک فرایند در تراز کاربر درگیر پیچیدگی‌ها و گوناگونی سخت‌افزاری نمی‌شود. برای نمونه، در نوشتن روی یک فایل، فرایند کاری ندارد که «۰» و «۱»‌های فایل کجای هارد جای می‌گیرند، یا آن فایل روی هارد است یا روی یک سی‌دی و اینها هر کدام چه ساختاری دارند. چنین چیزهایی را هسته رسیدگی می‌کند.

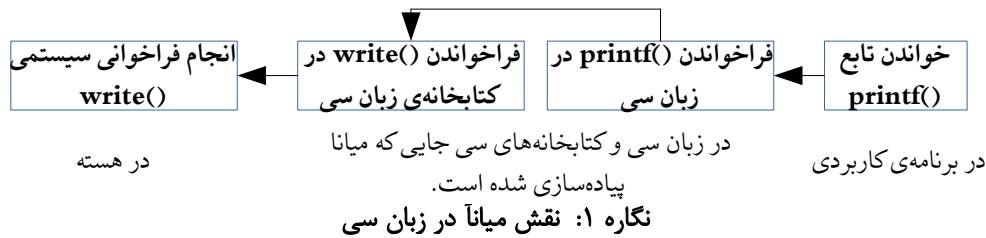
برخی سیستم‌عامل‌ها کاری به چگونگی دسترسی فرایندهای دیگر به دستگاه‌ها ندارند. در این سیستم‌عامل‌ها باید به دو چالش «جنگ میان فرایندها برای به‌دست‌گرفتن دستگاه‌ها» و نیز «بن‌بست^{۲۱}» رسیدگی شود.

19 hardware drivers

۲۰ در اینجا، کاربر همان برنامه‌ی کاربردی (و نه هسته) است.

21 deadlock

اگر همه‌ی فرآیندها می‌توانستند خودسرانه و بی‌برنامه‌ریزی با سخت‌افزار کار کنند، ناهنجاری پیش می‌آمد و نمی‌توانستیم به ساختارهای چندکاره دست یابیم. برنامه‌هایی که شما در تراز کاربر می‌نویسید با یک میان‌آه کار می‌کند که درون یک زبان برنامه‌نویسی، مانند سی، گنجانده شده است. در لینوکس، میان‌آه به زبان سی در دسترس است و می‌توان از کتابخانه‌های آن بهره برد. برای نمونه، روند درخواست فراخوان سیستمی `write` را در برنامه‌های نوشته شده به زبان سی در نگاره زیر می‌بینید:



میان‌آه در لینوکس بر پایه‌ی استانداردهی پوزیکس نوشته می‌شوند. در لینوکس، میان‌آه به زبان سی است و دیگر زبان‌ها نیز با نوشتن لایه‌ای به دورش، از همان بهره می‌برند.

چرا این کار انجام می‌شود؟ به زبان دیگر چرا برنامه‌نویس خودش با فراخوانی‌ها بی‌میانجی کار نمی‌کند؟ پاسخ این است که برای برنامه‌نویس کاربردی دانستن فراخوانی به کار رفته در یک دستور یا کار ارزشی ندارد و تنها خودِ کار و برآیند آن ارزشمند است.

از سوی دیگر، هسته تنها فراخوانی‌های سیستمی را می‌شناسد و نگران این نیست که کدام کتابخانه یا برنامه می‌خواهد یک فراخوانی را به کار گیرد. ولی، بهتر است فراخوانی‌های هسته باید چنان فراگیر باشند تا بتوانند هرگونه درخواستی را به خوبی پاسخ گویند.

۱.۶ سازوکار دسترسی به فراخوانی‌ها

فراخوانی‌های سیستمی که از راه تابع‌های کتابخانه‌ی زبان «سی» در دسترس هستند شاید آرگومان ورودی نیز داشته باشند. برخی از آنها ساختار را دستکاری می‌کنند، مانند نوشتن در یک پرونده، و برخی تنها داده‌هایی را برمی‌گردانند مانند `getpid()` که شناسه‌ی فرایند را می‌دهد. ساختار فراخوانی `getpid()` اینگونه است:

```

1 SYSCALL_DEFINE0(getpid)
2 {
3     return task_tgid_vnr(current); //returns current tgid
4 }
    
```

همانگونه که می‌بینید پیاده‌سازی در اینجا انجام نشده است. هسته باید رفتار فراخوانی را از روی یک پیاده‌سازی فراهم سازد. ساختار بازشده‌ی آن این است:

asmlinkage long sys_getpid(void)

این تنها شناساندن فراخوانی است که دارای بخش‌های زیر است:

asmlinkage: باید در همه‌ی فراخوانی‌های سیستمی نوشته شود و به کامپایل کننده می‌گوید که در پشته تنها به دنبال آرگومانهای این تابع بگردد.

long: اندازه‌ی آن‌چه که فراخوانی برمی‌گرداند دارای الگوی long است؛ برای هماهنگی در سیستم‌های ۶۴ و ۳۲ بیتی.

sys_getpid(): نام هر فراخوانی در هسته لینوکس باید با sys_ آغاز شود؛ برای نمونه، فراخوانی xyz در لینوکس، در هسته‌ی آن با تابع sys_xyz() پیاده‌سازی می‌شود.

هر فراخوانی سیستمی دارای یک شماره‌ی یکتاست. یک فرایند تراز کاربر با فرستادن همین شماره به هسته، یک فراخوانی را درخواست می‌کند. نمی‌توان این شماره را جایگزین یا جابجا کرد. نمی‌توان آن را پاک کرد، وگرنه برنامه‌های کامپایل شده که این فراخوانی را دارند کار نمی‌کنند. هسته فهرست فراخوانی‌ها را در جدولی که در فایل sys_call_table است نگه می‌دارد. در ساختار x86_64 این فایل در نشانی arch/x86/kernel/syscall.64.c نگهداری می‌شود. در این جدول به هر فراخوانی یک شماره یکتا داده شده است.

برنامه‌ها خود نمی‌توانند دستورهای تراز هسته را اجرا کنند. پس برای اینکه بتوانند از فراخوانی‌های سیستمی بهره بگیرند با کمک یک گسل (ایست، وقفه) نرم‌افزاری، پردازنده به تراز هسته می‌رود. سپس، در تراز هسته، خودِ هسته به انجام فراخوانی درخواستی از سوی برنامه کاربردی می‌پردازد و پاسخش را برمی‌گرداند.

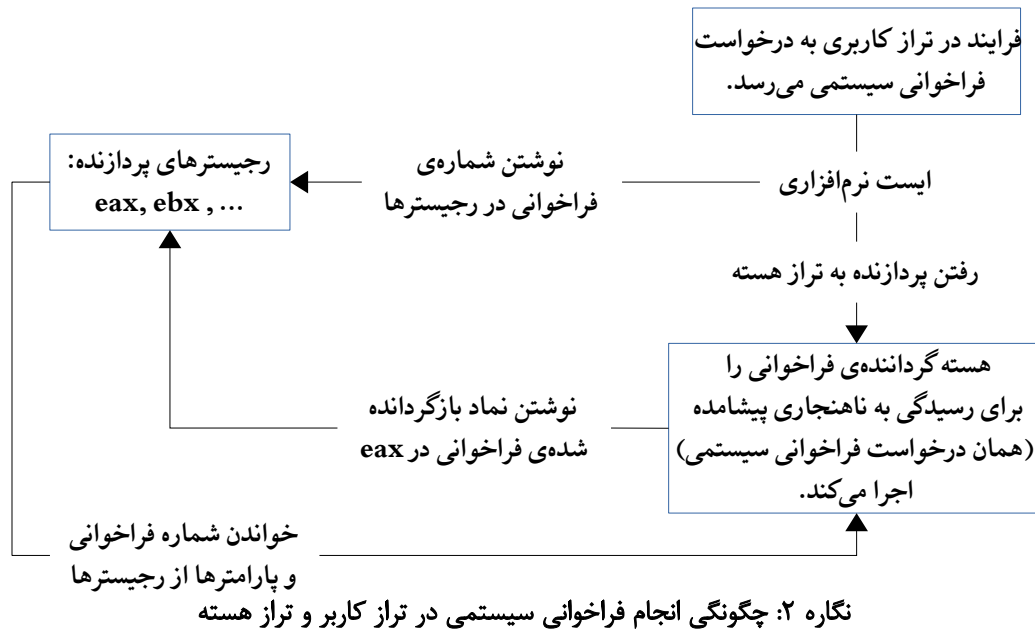
ایست یا گسل نرم‌افزاری یک «ناهنجاری» در زمان انجام پیش می‌آورد تا پردازنده به تراز هسته برود تا در آنجا هسته به این ناهنجاری رسیدگی کند. این رسیدگی همان انجام فراخوانی سیستمی است که کاربر درخواست کرده بود. ایست نرم‌افزاری در ساختار x86 با شماره‌ی ۱۲۸ است که با دستور int \$0x80 پدید می‌آید. این دستور پردازنده را به تراز هسته برده و «ناهنجاری زمان اجرای» ردیف ۱۲۸ که همان گرداننده‌ی فراخوانی^{۲۳} سیستمی است بررسی می‌شود. کوتاه اینکه، تراز کاربری با پدیدآوردن یک ناهنجاری زمان اجرا (یا یک دام دست‌ساز برای برنامه) به درون تراز هسته می‌افتد تا در آنجا به فراخوانی سیستمی رسیدگی شود.

۱.۶.۱ کدام فراخوانی باید انجام شود؟ شماره‌ی فراخوانی چگونه به دست هسته می‌رسد؟

در x86 برنامه‌ی کاربردی در تراز کاربر شماره‌ی فراخوانی درخواستی خود را در رجیستر eax پردازنده می‌نویسد. گرداننده‌ی فراخوانی، در تراز هسته، نخست این حافظه را می‌خواند، شماره را برای بررسی درستی به تابع () system_call می‌دهد. اگر این شماره بزرگتر یا برابر با NR_syscalls بود، هشدار _ENOSYS (نشان دهنده‌ی این که چنین فراخوانی ساخته نشده است) را برمی‌گرداند وگرنه فراخوانی را صدا می‌زند:

```
call *sys_call_table(, %rax, 8)
```

برای فرستادن پارامترهای فراخوانی نیز نرم افزار کاربردی از رجیسترهای دیگر پردازنده بهره می برد. هسته پاسخ انجام فراخوانی را با کمک رجیستر `eax` به تراز کاربر می رساند. (نگاره ۲)



۱.۷ ردگیری فراخوانی‌ها

هر دستور یا برنامه‌ای برای انجام کار خودش به چندین فراخوانی سیستمی (پایین رده) نیاز دارد. برای این که دریابیم که یک برنامه از چه فراخوانی‌هایی کمک گرفته است می توان از دستور `strace` در لینوکس بهره برد. شیوه‌ی کار با این دستور ساده و مانند زیر است:

```
$ strace <دستور>
```

برای نمونه می‌خواهیم بینیم ابزار `date` که زمان را نشان می‌دهد با هر بار انجام چه فراخوانی‌هایی را به کار می‌گیرد، پس:

```
$ strace date
```

دستآورد این دستور در نگاره ۳ دیده می‌شود.

در این جا هر خط نماینده‌ی یک فراخوانی سیستمی به کاررفته در زمان اجرای دستور `date` است. کارکرد برخی از آن‌ها را می‌توانید در نگاره ۷ (پایان بخش) پیدا کنید.


```

1 #include <linux/kernel.h>
2 #include <linux/syscalls.h>
3
4 SYSCALL_DEFINE0(oslab)
5 {
6     printk("\nThis is OSLAB's system call.\n");
7     return 0;
8 }

```

در این تکه برنامه از تابع `printk` بهره برده ایم. این تابع رشته ی ورودی خود را در فایل `log` (رویدادهای) هسته می نویسد. همچنین، می بینید که نام فراخوانی آرگومان ورودی `SYSCALL_DEFINE0`. عدد صفر نشان دهنده ی این است که آرگومان ورودی ندارد.

هشدار: در هسته ما به دستور `printf` دسترسی نداریم! از اینرو تنها می توانیم در پرونده ی رویدادهای هسته با کمک فرمان `printk` بنویسیم.

۱.۸.۲ جایگذاری فراخوانی پایین رده در سیستم عامل

۱. در پوشه ای که از نافشرده سازی هسته ی بارگیری شده در آزمایش پیشین (کامپایل هسته) به دست آمد، پوشه ای تازه برای فراخوانی های سیستمی خودمان می سازیم (ما نام آن را `mysyscalls`) می نامیم.

```

$ cd linux-5.9.6/
$ mkdir mysyscalls

```

۲. سپس، فایل فراخوانی `oslab.c` را درون این پوشه می گذاریم.

۳. روند کار اینگونه است که فایل `oslab.c` در «هنگام کامپایل و ساخت هسته» کامپایل شود و برنامه ی هسته به آن دسترسی داشته باشد. برای این که در هنگام کامپایل، بدانیم که این فایل باید چگونه ساخته (`make`) شود، برای آن یک `Makefile` می سازیم.

```

$ cd mysyscalls
$ nano Makefile

```

در این فایل چنین چیزی می نویسیم.

```

GNU nano 4.8      Makefile
obj-y := oslab.o

```

نگاره ۴: پرونده ی `Makefile` برای فراخوانی سیستم `oslab.c`

۴. اکنون باید در فایل `Makefile` اصلی هسته، که هسته برپایه ی آن ساخته می شود، به سازنده ی هسته نشان دهیم که فراخوانی سیستم تازه ی ما و فایل سازنده ی (`Makefile`) آن کجاست. پس، فایل `Makefile` هسته را باز می کنیم. رشته ی «`core-y`» را جستجو می کنیم و نشانی پوشه ی `mysyscalls` را به فهرست پوشه های روبروی آن می افزاییم. در دنباله ی دستورهای گام ۳ این دستورها را می توانیم به کار ببریم:

```

$ cd ..
$ nano Makefile

```

پس از جستجوی «core-y» در خطی مانند زیر نشانی پوشه را می‌افزاییم (در ویرایشگر nano می‌توانید رشته‌ای را با کلید ترکیبی ctrl+w جستجو کنید). فایل را ذخیره می‌کنیم و می‌بندیم.

```
ifeq ($(KBUILD_EXTMOD),)
core-y += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ mysyscalls/
vmlinux-dirs := $(patsubst %/,,$(filter %/, \
```

۵. افزودن فراخوانی به جدول فراخوانی‌های معماری؛ این جدول درون یک فایل است. در این فایل، در هر خط یک فراخوانی به هسته شناسانده شده و یک شماره به آن داده شده است. شماره‌گذاری از «۰» است. برای هر معماری که از فراخوانی پشتیبانی می‌کند، این جایگذاری باید انجام شود. برای نمونه، در سیستم ما که x86_64 است ما فراخوانی را به جدول فراخوانی‌ها برای معماری ۶۴ بیت می‌افزاییم. پس از ذخیره و بستن فایل Makefile در گام پیشین می‌توانید دستورهای زیر را انجام دهید.

```
$ nano arch/x86/entry/syscalls/syscall_64.tbl
```

در فایل‌ای که باز می‌شود، پس از فراخوانی‌های سیستمی ویژه‌ی ۶۴ بیت و پیش از فراخوانی‌های ویژه‌ی ۳۲ بیت باید فراخوانی تازه را به فهرست بیافزاییم. چیزی مانند نگاره ۵.

424	common	pidfd_send_signal	sys_pidfd_send_signal
425	common	io_uring_setup	sys_io_uring_setup
426	common	io_uring_enter	sys_io_uring_enter
427	common	io_uring_register	sys_io_uring_register
428	common	open_tree	sys_open_tree
429	common	move_mount	sys_move_mount
430	common	fsopen	sys_fsopen
431	common	fsconfig	sys_fsconfig
432	common	fsmount	sys_fsmount
433	common	fspick	sys_fspick
434	common	pidfd_open	sys_pidfd_open
435	common	clone3	sys_clone3
436	common	close_range	sys_close_range
437	common	openat2	sys_openat2
438	common	pidfd_getfd	sys_pidfd_getfd
439	common	faccessat2	sys_faccessat2
440	common	oslab	sys_oslab
#			
# x32-specific system call numbers start at 512 to avoid cache			

نگاره ۵: فایل syscall_64.tbl و افزودن نام و شماره‌ی فراخوانی تازه به فهرست فراخوانی‌ها.

شماره‌ی فراخوانی تازه‌ی ما ۴۴۰ شده است. در ویرایش‌های دیگر هسته این شناسه می‌تواند چیز دیگری باشد.

۶. اکنون باید دسته‌ی فراخوانی تازه را در «سربرگ فراخوانی‌های لینوکس» بگذاریم. این همان سربرگی است که در زبان سی بالای برنامه می‌افزاییم تا این فراخوانی در دسترس باشد. در دنباله‌ی دستورهای گام پیشین دستور زیر را می‌نویسیم.

```
$ nano include/linux/syscalls.h
```

در پایان این پرونده، و پیش از دستور #endif دسته‌ی دستورکار خود را می‌افزاییم (نگاره ۶).

```
int __sys_getsockopt(int fd, int level, int optname, char __user *optval,
                    int __user *optlen);
int __sys_setsockopt(int fd, int level, int optname, char __user *optval,
                    int optlen);

asm linkage long sys_oslab(void);

#endif
```

نگاره ۶: فایل `include/linux/syscalls.h` و افزودن دستینه‌ی فراخوانی سیستمی به پایان آن.

در اینجا جایگذاری فراخوانی در میان پرونده‌های هسته پایان یافت. اکنون باید این هسته را مانند آنچه در «دستورکار پیشین» انجام شد «می‌سازیم».

این هسته را بالا بیاورید. فرآیندهای تراز کاربر می‌تواند از فراخوانی `oslab()` بهره ببرند. این روند ساخت یک فراخوانی بود. ما همچنین می‌توانیم فراخوانی‌های کنونی هسته را نیز دستکاری کنیم. برای این کار تنها برنامه و کدها را دستکاری کنیم و نیازی به جایگذاری فایل‌ها نیست.

۱.۸.۳ دسترسی به فراخوانی در تراز کاربر

پس از بالا آمدن هسته‌ی تازه، برنامه‌ی ساده‌ای می‌نویسیم تا از فراخوانی سیستمی `oslab` که در هسته گنجاندیم در آن بهره ببریم. ما برنامه‌ی ساده‌ی زیر (`test_added_syscall.c`) را به کار بردیم. به سرب‌گ‌هایی که در این برنامه خوانده‌ایم نگاه کنید. نام‌های آن‌ها آشنا نیست؟ همچنین در اینجا فراخوانی را با شماره‌ی آن، «۴۴۰»، فراخوانده‌ایم.

```
GNU nano 4.8      test_added_syscall.c      Modified
#include <stdio.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>

int main(){
    long int syscall_return_value = syscall(440);
    printf("The output of oslab syscall: %ld\n", syscall_return_value);
    return 0;
}
```

کامپایل و اجرای این برنامه خروجی زیر را می‌دهد.

```
The output of oslab syscall: 0
```

می‌بینید که مقدار بازگشتی فراخوانی، که «۰» است، نشان داده می‌شود. ولی، فراخوانی ما کار دیگری انجام می‌داد و آن هم نوشتن رشته‌ای در فایل `log` هسته بود (با دستور `printk`). برای اینکه ببینیم در فایل `log` هسته چه رخ داده است باید دستور زیر را اجرا کنیم.

```
$ dmesg
```

که فایل را نمایش می‌دهد. می‌بینید که در پایان این فایل آنچه فراخوانی ما، با دو بار اجرا، نوشته است افزوده شده است.

```

dDevice= 1.00
[ 646.323945] usb 2-1: New USB device strings: Mfr=1, Product=3, SerialNumber=
0
[ 646.323948] usb 2-1: Product: USB Tablet
[ 646.323950] usb 2-1: Manufacturer: VirtualBox
[ 646.545507] input: VirtualBox USB Tablet as /devices/pci0000:00/0000:00:06.0
/usb2/2-1/2-1:1.0/0003:80EE:0021.0002/input/input7
[ 646.606203] hid-generic 0003:80EE:0021.0002: input,hidraw0: USB HID v1.10 Mo
use [VirtualBox USB Tablet] on usb-0000:00:06.0-1/input0
[ 649.603687] e1000: enp0s3 NIC Link is Up 1000 Mbps Full Duplex, Flow Control
: RX
[ 1245.770447]
This is OSLAB's system call.
[ 1463.555994]
This is OSLAB's system call.

```

اگر می‌خواهید درون این فایل رویدادنگاری پاک شود، می‌توانید از گزینه‌ی clear آن اینگونه بهره ببرید:

```
$ dmesg --clear
```

۱.۹ دستورکار

۱. با فراخوانی fork پیش‌تر کار کرده‌اید. این فراخوانی چه آرگومان‌های ورودی دریافت می‌کند و چه بازگشتی‌هایی دارد؟ اگر پس از انجام این دستور، فرایند فرزند ساخته نشد، این فراخوانی چه چیزی برمی‌گرداند؟

• در سیستم عامل ویندوز چه فراخوانی‌ای کار fork را انجام می‌دهد؟

۲. چقدر تا از فراخوانی‌های سیستمی خانواده‌ی exec را بررسی کنید.

۳. یک برنامه با چنین ساختاری داریم. به جای «؟» یک شماره گذاشته می‌شود و به برنامه‌ی فراخوانده چیزی برمی‌گرداند. چند نمونه از این شماره‌ها پیدا کنید و بگویید این‌ها چه هستند؟ هر کدام نشان از چه دارند؟

```

int main(){
    return ?;
}

```

۴. آنچه که در فایل Makefile برای فراخوانی سیستمی oslab.c نوشتیم را توضیح دهید. منظور از بخش‌های گوناگون آن چیست؟

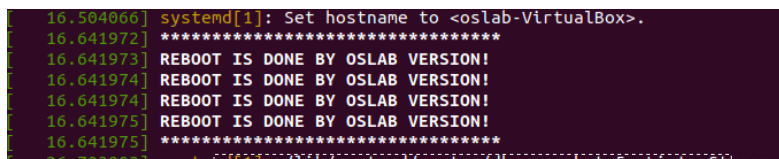
۵. فراخوانی سیستمی‌ای «با نام خودتان» به هسته‌ی لینوکس بیافزایید که کارش نوشتن نام شما در فایل log (رویداد) هسته باشد. در لینوکس با این هسته‌ی کامپایل شده، برنامه‌ی ساده‌ای بنویسید که کاربرد این فراخوانی را نشان دهد (اگر نام شما sohrab rostami است نام فراخوانی شما باید sohrabrostami باشد).

۶. برنامه‌ی زیر چه کاری انجام می‌دهد؟ این برنامه کار چه برنامه‌ی دیگری را شبیه‌سازی می‌کند؟

```
1 #include<stdio.h>
2 #include<sys/types.h>
3 #include <unistd.h>
4 #include<stdlib.h>
5
6 void main(){
7     char cmd[100];
8     pid_t fake_shell_pid, copied_shell_pid;
9     int status;
10    //fake_shell_pid = fork();
11    while(1){
12        printf("\noslab@scu:$ Enter a Command : ");
13        scanf("%s",cmd);
14        //gets(cmd);
15        printf("Original Shell PID: %d\n", getpid());
16        copied_shell_pid = fork();
17        if(copied_shell_pid < 0)
18            printf("Failed to run the command.\n");
19        else if (copied_shell_pid > 0)
20            waitpid(-1, &status, 0);
21        else{
22            printf("Copied Shell PID: %d\n", getpid());
23            char *args[]={cmd, NULL};
24            execvp(args[0],args);
25        }
26    }
27 }
```

برنامه ۱

۷. فراخوانی سیستمی که برای بازراه‌اندازی (reboot) سیستم عامل است را به‌گونه‌ای دست‌کاری کنید که: هر بار که سیستم عامل را بازراه‌اندازی (reboot یا restart) می‌کنیم نوشته‌ای ویژه مانند زیر در فایل لاگ (log) لینوکس نوشته شود. می‌بینید که در فایل رویداد زیر، دو خط دارای ستاره (*) و رشته‌های میان آن‌ها با هر بار بازراه‌اندازی لینوکس ما نوشته می‌شود.



(راهنمایی: فراخوانی سیستمی reboot را در فایل linux/kernel/reboot.c از پوشه‌ی هسته‌ی بارگیری‌شده پیدا کنید. برای دیدن لاگ (رویدادها) پس از کامپایل نیز از دستور dmesg کمک بگیرید.)

۸. با دستور strace آشنا شده‌اید. با کمک این دستور و نیز فهرست داده‌شده در نگاره ۷ برخی از فراخوانی‌هایی که دستور date از آن‌ها برای انجام کار خود کمک می‌گیرد را گزارش کنید.

۹. برنامه‌ای به نام copy_file.c بنویسید. در این برنامه داده‌ها را از پرونده‌ای به نام src.txt «بخوانید». سپس، فایل دیگری به نام dst.txt «بسازید» و داده‌های خوانده‌شده را به آن «بنویسید». در پایان هر دو فایل باید «بسته» شوند. با کمک دستور strace نشان دهید که چه فراخوانی‌هایی را برنامه‌ی شما به کار می‌برد. همچنین، بگویید کارهایی که در «آ» آمده است با کدام فراخوانی‌ها انجام گرفته است.

۱.۱۰ پرسش‌های ژرف

۱. آیا می‌توان سیستم عاملی داشت که فراخوانی‌های سیستمی نداشته باشد. به‌زبان دیگر، همه‌ی برنامه‌ها در تراز کاربر کار کنند؟ چالش‌های چنین سیستم عاملی چه هستند؟

Process management	<p> pid = fork() pid = waitpid(pid, &statloc, opts) s = wait(&status) s = execve(name, argv, envp) exit(status) size = brk(addr) pid = getpid() pid = getpgrp() pid = setsid() l = ptrace(req, pid, addr, data) </p>	<p> Create a child process identical to the parent Wait for a child to terminate Old version of waitpid Replace a process core image Terminate process execution and return status Set the size of the data segment Return the caller's process id Return the id of the caller's process group Create a new session and return its proc. group id Used for debugging </p>
Signals	<p> s = sigaction(sig, &act, &oldact) s = sigreturn(&context) s = sigprocmask(how, &set, &old) s = sigpending(set) s = sigsuspend(sigmask) s = kill(pid, sig) residual = alarm(seconds) s = pause() </p>	<p> Define action to take on signals Return from a signal Examine or change the signal mask Get the set of blocked signals Replace the signal mask and suspend the process Send a signal to a process Set the alarm clock Suspend the caller until the next signal </p>
File Management	<p> fd = creat(name, mode) fd = mknod(name, mode, addr) fd = open(file, how, ...) s = close(fd) n = read(fd, buffer, nbytes) n = write(fd, buffer, nbytes) pos = lseek(fd, offset, whence) s = stat(name, &buf) s = fstat(fd, &buf) fd = dup(fd) s = pipe(&fd[0]) s = ioctl(fd, request, argp) s = access(name, amode) s = rename(old, new) s = fcntl(fd, cmd, ...) </p>	<p> Obsolete way to create a new file Create a regular, special, or directory i-node Open a file for reading, writing or both Close an open file Read data from a file into a buffer Write data from a buffer into a file Move the file pointer Get a file's status information Get a file's status information Allocate a new file descriptor for an open file Create a pipe Perform special operations on a file Check a file's accessibility Give a file a new name File locking and other operations </p>
Dir. & File System Mgt.	<p> s = mkdir(name, mode) s = rmdir(name) s = link(name1, name2) s = unlink(name) s = mount(special, name, flag) s = umount(special) s = sync() s = chdir(dirname) s = chroot(dirname) </p>	<p> Create a new directory Remove an empty directory Create a new entry, name2, pointing to name1 Remove a directory entry Mount a file system Unmount a file system Flush all cached blocks to the disk Change the working directory Change the root directory </p>
Protection	<p> s = chmod(name, mode) uid = getuid() gid = getgid() s = setuid(uid) s = setgid(gid) s = chown(name, owner, group) oldmask = umask(complmode) </p>	<p> Change a file's protection bits Get the caller's uid Get the caller's gid Set the caller's uid Set the caller's gid Change a file's owner and group Change the mode mask </p>
Time Management	<p> seconds = time(&seconds) s = stime(tp) s = utime(file, timep) s = times(buffer) </p>	<p> Get the elapsed time since Jan. 1, 1970 Set the elapsed time since Jan. 1, 1970 Set a file's "last access" time Get the user and system times used so far </p>

Figure 1-9. The main MINIX system calls. *fd* is a file descriptor; *n* is a byte count.

نگاره ۷: فراخوانی‌های پرکاربرد لینوکس