به نام خدا

گزارش اول آزمایشگاه سیستم عامل

سید علیرضا میرشفیعی - ۸۱۰۱۰۱۵۳۲
محمد تقی زاده - ۸۱۰۱۰۲۵۸۹
محمد صدرا عباسی - ۸۱۰۱۰۱۴۶۹

(1

سه وظیفه اصلی سیستم عامل:
۱. واسط بین نرم افزار و سخت افزار : سیستم عامل پلی میان برنامه های کاربردی و کاربران در لایه بالا و سخت افزار در لایه پایین است.
۲. مدیریت منابع سخت افزاری : اشتراک گذاری و تخصیص بهینه منابع.
۳. مدیریت نیازهای نرم افزاری و تعامل برنامه ها : پنهان سازی پیچیدگی های سخت افزار، ارائه سرویس های سطح بالا و کنترل ارتباط بین برنامه ها.

(2

خیر، وجود سیستم عامل در تمام دستگاه‌ها الزامی نیست.

چون دستگاه‌های بسیار ساده با کاربردی واحد و مشخص، نیازی به سیستم عامل ندارند. برای مثال، یک دستگاه با مدار مجتمع خاص کاربردی (ASIC) وظایف خود را بدون داشتن سیستم عامل انجام می‌دهد. کامپیوترهای تعبیه‌شده در برخی لوازم خانگی ساده مانند یک مایکروفر، تنها یک برنامه کنترلی ثابت (سفت‌افزار) دارند که مستقیماً سخت‌افزار را برای انجام یک کار مشخص مدیریت می‌کند و فاقد سیستم عامل به معنای رایج آن هستند.

در چه شرایطی استفاده از سیستم عامل لازم است؟

زیرا راه حلی منطقی برای ایجاد یک سیستم محاسباتی عملی و قابل استفاده ارائه می‌دهند. نیاز به سیستم عامل در شرایط زیر آشکار می‌شود:

۱. مدیریت منابع مشترک

۲. فراهم کردن رابط کاربری و سادگی استفاده

۳. ایجاد بستری برای اجرای برنامه‌های کاربردی

(3

از ساختار یکپارچه مشابه Unix استفاده می کند.
دلایل:

اجرای تمام سرویس‌ها در فضای هسته: در سیستم عامل xv6، تمامی سرویس‌های حیاتی در فضای هسته اجرا می‌شوند.

رابط فراخوانی سیستمی (System Call Interface): واسطی که به برنامه‌های کاربردی اجازه می‌دهد تا از خدمات هسته استفاده کنند. این ساختار دقیقاً با تعریف هسته یکپارچه مطابقت دارد که در آن حجم عظیمی از کارکردها در یک فضای آدرس منفرد ترکیب می‌شود.
عدم وجود ماژولاریتی در سطح معماری اصلی

(4

سیستم عامل xv6 یک سیستم عامل چند وظیفه است

5)

تفاوت اساسی بین یک برنامه و یک پردازه در **فعال یا منفعل بودن** آن‌هاست. به طور خلاصه، برنامه یک موجودیت منفعل است، در حالی که پردازه یک موجودیت فعال و در حال اجراست. یک برنامه می‌تواند مبنای ایجاد چندین پردازه باشد. برای مثال، چندین کاربر مختلف می‌توانند کپی‌های متفاوتی از یک برنامه مرورگر وب را اجرا کنند که هر کدام یک پردازه مجزا ایجاد می‌کند. در این حالت، بخش کد (دستورالعمل‌ها) بین این پردازه‌ها یکسان است، اما هر پردازه بخش‌های داده، هیپ و پشته مخصوص به خود را دارد.

ویژگی	**برنامه (Program)**	**پردازه (Process)**
ماهیت	موجودیت منفعل و ایستا	موجودیت فعال و پویا
محل ذخیره‌سازی	فایل بر روی حافظه ثانویه (دیسک)	در حافظه اصلی در حین اجرا
اجزا	مجموعه‌ای از دستورالعمل‌ها	شمارنده برنامه، ثبات‌ها، پشته، هیپ و منابع سیستم
طول عمر	نامحدود (تا زمانی که حذف نشود)	محدود به زمان اجرای برنامه

6)

در سیستم عامل xv6، ساختار یک پردازه همانند دیگر سیستم‌عامل‌های مبتنی بر یونیکس، از چندین بخش اصلی در حافظه تشکیل شده است. این ساختار به پردازه اجازه می‌دهد تا کد خود را اجرا کرده و داده‌هایش را مدیریت کند. این بخش‌ها عبارتند از:

بخش متن (Text Section): این بخش شامل کد اجرایی برنامه است که دستورالعمل‌های پردازنده را در بر می‌گیرد. بخش داده‌ها (Data Section): متغیرهای سراسری برنامه در این قسمت ذخیره می‌شوند. هیپ (Heap) و پشته (Stack)

این ساختار باعث می‌شود که هر پردازه یک فضای آدرس منطقی مجزا و محافظت‌شده داشته باشد و بتواند به صورت مستقل از سایر پردازه‌ها اجرا شود

نحوه اختصاص پردازنده به پردازه‌ها در xv6:

سیستم عامل xv6 به عنوان یک سیستم چند وظیفه، پردازنده را با استفاده از یک الگوریتم زمان‌بندی نوبت-چرخشی (Round-Robin) به پردازه‌های مختلف اختصاص می‌دهد. این روش یکی از ساده‌ترین و منصفانه‌ترین الگوریتم‌های زمان‌بندی است.

کلیت:

1. صف آماده (Ready Queue): سیستم عامل لیستی از تمام پردازه‌هایی که آماده اجرا هستند (در حالت runnable) را نگهداری می‌کند.
2. انتخاب پردازه: زمان‌بند (Scheduler) به ترتیب در میان این لیست حرکت کرده و پردازنده را برای یک دوره زمانی کوتاه و مشخص به نام برش زمانی (Time Slice) به هر پردازه اختصاص می‌دهد.
3. جابجایی (Context Switch): پس از اتمام برش زمانی یک پردازه، یا اگر پردازه به صورت داوطلبانه پردازنده را رها کند (مثلاً برای انتظار یک عملیات ورودی/خروجی)، یک جابجایی متن (Context Switch) رخ می‌دهند. در این فرآیند، حالت فعلی پردازه (شامل مقادیر ثبات‌ها و شمارنده برنامه) در بلوک کنترل پردازه (PCB) آن ذخیره شده و حالت پردازه بعدی از لیست، برای اجرا بارگذاری خواهد شد.
4. تکرار چرخه: این چرخه به طور مداوم تکرار می‌شود و پردازنده به نوبت در اختیار تمام پردازه‌های آماده قرار می‌گیرد. این جابجایی سریع بین پردازه‌ها این تصور را برای کاربر ایجاد می‌کند که تمام برنامه‌ها به صورت همزمان در حال اجرا هستند.

این روش ساده تضمین می‌کند که هیچ پردازه‌ای برای مدت طولانی از دسترسی به پردازنده محروم نمی‌شود

(7

در سیستم‌عامل‌های مبتنی بر یونیکس مانند xv6، یک اصل طراحی مهم وجود دارد و آن این است که با همه چیز تا حد امکان مانند یک فایل رفتار شود. این اصل به یکپارچگی و سادگی سیستم کمک می‌کند. برای مدیریت این "فایل‌ها" (که می‌توانند فایل‌های واقعی روی دیسک، دستگاه‌هایی مانند کیبورد، یا کانال‌های ارتباطی باشند)، سیستم از یک مفهوم کلیدی به نام **توصیف‌گر فایل (File Descriptor)** استفاده می‌کند.

وقتی یک پردازه، فایلی را باز می‌کند، هسته به جای اینکه پردازه را مجبور کند هر بار با نام طولانی فایل کار کند، یک عدد صحیح کوچک و ساده به آن اختصاص می‌دهد. این عدد همان توصیف‌گر فایل است. در واقع، این عدد یک شناسه برای فایل باز شده در جدول مخصوصی است که هسته برای هر پردازه نگهداری می‌کند. از آن پس، تمام کارهایی که پردازه می‌خواهد روی آن فایل انجام دهد، مانند خواندن و نوشتن، با استفاده از همین عدد ساده انجام می‌شود. این روش کار را بسیار سریع‌تر و راحت‌تر می‌کند.

این ایده به ارتباط بین پردازه‌ها نیز گسترش پیدا کرده است. یکی از اولین و مهم‌ترین مکانیزم‌ها برای این کار، **لوله یا Pipe** است. pipe در واقع یک کانال یک‌طرفه است که هسته بین دو پردازه ایجاد می‌کند. وقتی یک pipe ساخته می‌شود، سیستم دو توصیف‌گر فایل به پردازه می‌دهد: یکی برای نوشتن در pipe و دیگری برای خواندن از آن. حالا یک پردازه می‌تواند داده‌ها را در یک سر pipe بنویسد و پردازه دیگر آن‌ها را از سر دیگر pipe بخواند.

معمول‌ترین کاربرد pipe برای اتصال خروجی یک برنامه به ورودی برنامه‌ای دیگر است تا یک **خط لوله (pipeline)** از دستورات ایجاد شود. برای مثال، وقتی در خط فرمان می‌نویسیم ls | more، خروجی دستور ls که لیست فایل‌هاست، به جای چاپ شدن روی صفحه، مستقیماً به ورودی دستور more فرستاده می‌شود. این کار به شما اجازه می‌دهد تا یک لیست طولانی را به صورت صفحه به صفحه مشاهده کنید، بدون اینکه نیاز به ذخیره خروجی ls در یک فایل موقت داشته باشید. این قابلیت، یکی از قدرتمندترین ویژگی‌های محیط‌های مبتنی بر یونیکس است.

(8

در سیستم عامل xv6، فراخوانی‌های سیستمی `fork()` و `exec()` دو ابزار بنیادی برای ایجاد و مدیریت پردازه‌های جدید هستند، اما هر کدام وظیفه‌ای کاملاً متفاوت را انجام می‌دهند.

فراخوانی `fork()` در واقع یک "کپی" یا "کلون" از پردازه فعلی ایجاد می‌کند. وقتی یک پردازه `fork()` را صدا می‌زند، سیستم عامل یک پردازه جدید (فرزند) می‌سازد که تقریباً یک کپی دقیق از پردازه والد است. این کپی شامل فضای حافظه، داده‌ها و توصیف‌گرهای فایل والد می‌شود. نکته کلیدی این است که پس از `fork()`، هر دو پردازه (والد و فرزند) اجرای خود را از دستور بعدی ادامه می‌دهند. تنها راه تشخیص آن‌ها از یکدیگر، مقدار بازگشتی fork است: در پردازه والد، این مقدار برابر با شناسه پردازه (PID) فرزند است و در پردازه فرزند، مقدار آن صفر است.

از طرف دیگر، فراخوانی exec کار کاملاً متفاوتی انجام می‌دهد. این فراخوانی، فضای حافظه پردازه فعلی را به طور کامل با یک برنامه جدید جایگزین می‌کند. به عبارت دیگر، `exec()` یک برنامه اجرایی را از دیسک بارگذاری کرده و آن را در همان پردازه فعلی اجرا می‌کند. شناسه پردازه تغییر نمی‌کند، اما برنامه‌ای که در حال اجرا بود کاملاً از بین رفته و برنامه جدید جای آن را می‌گیرد. اگر `exec()` با موفقیت اجرا شود، هرگز به برنامه‌ای که آن را فراخوانی کرده باز نمی‌گردد.

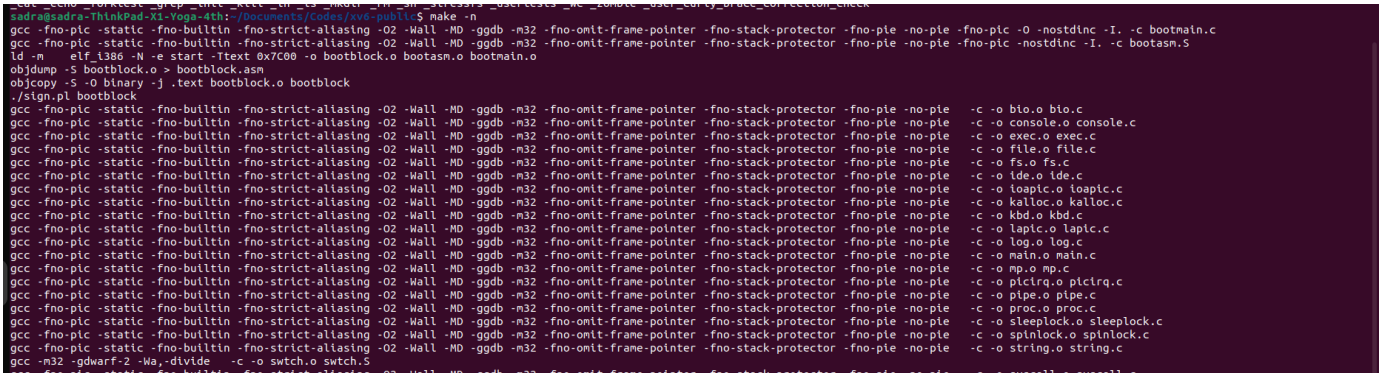
مزیت ادغام نکردن fork() و exec()

از نظر طراحی، جدا نگه داشتن این دو عملیات یک مزیت بسیار بزرگ دارد و آن **انعطاف‌پذیری** است. فاصله زمانی کوتاهی که بین fork و exec وجود دارد، به پردازه والد این فرصت را می‌دهد تا محیط اجرای فرزند را قبل از اینکه برنامه جدید شروع به کار کند، دستکاری و آماده‌سازی نماید. این انعطاف‌پذیری برای پیاده‌سازی ویژگی‌های قدرتمند خط فرمان یونیکس، مانند **تغییر مسیر ورودی/خروجی (I/O Redirection)** و **لوله (Pipe)**، حیاتی است..

اگر `fork()` و `exec()` در یک فراخوانی واحد ادغام می‌شدند، این فرصت برای دستکاری محیط فرزند از بین می‌رفت و پیاده‌سازی این ویژگی‌های کلیدی بسیار دشوار یا غیرممکن می‌شد.

(9

دستور make-n فقط دستورات کامپایل و لینک کردن را نمایش می دهد اما هیچ فایلی ایجاد نمی کند. هدف نهایی این دستورات، ساخت img6.xv است که شامل کرنل و بوت لودر می شود. برای ساخت فایل نهایی کرنل از دستور make kernel استفاده می شود.



(10

۱. متغیر UPROGS
این متغیر لیست برنامه های کاربری (User Programs) را مشخص می کند که باید برای اجرا در XV6 کامپایل شوند.
۲. متغیر ULIB
این متغیر شامل کتابخانه های مورد نیاز برای برنامه های کاربری است.

(11

دستور **cd** (مخفف change directory یا تغییر پوشه) با دستورهای دیگری مثل **ls** (برای لیست کردن فایل‌ها) یا **cat** (برای نمایش محتوای فایل) فرق دارد. دستور cd باید مستقیماً در خود **پوسته** (shell) اجرا شود و نمی‌تواند مثل یک برنامه جداگانه اجرا شود.

دلیلش این است که دستور cd وظیفه دارد **پوشه فعلی** خودِ پوسته را عوض کند. اگر cd مثل برنامه‌های دیگر اجرا می‌شد، اتفاق زیر می‌افتاد:

1. پوسته یک **کپی از خودش** ایجاد می‌کرد (که به آن فرآیند فرزند یا child process می‌گوییم).
2. این کپی (فرآیند فرزند)، دستور cd را اجرا می‌کرد و پوشه فعلی خودش را تغییر می‌داد.
3. اما پوشه فعلی **پوسته اصلی** (فرآیند والد یا parent process) هیچ تغییری نمی‌کرد.

چون هدف ما تغییر پوشه فعلی پوسته اصلی است، دستور cd باید به عنوان بخشی داخلی از خود پوسته عمل کند تا بتواند مستقیماً حالت پوسته را تغییر دهد.

(12

طبق مستندات ارائه شده، محتوای سکتور اول (سکتور بوت) دیسک قابل بوت در سیستم‌عامل xv6، **بوت لودر** (Bootloader) است . این بوت لودر، کدی است که وظیفه دارد هسته سیستم‌عامل xv6 را از دیسک در حافظه بارگذاری کرده و سپس کنترل اجرای برنامه را به هسته منتقل کند. کد اصلی بوت لودر که در سکتور اول قرار می‌گیرد، از فایل bootasm.S مشتق شده است. BIOS پس از روشن شدن سیستم، این سکتور را در آدرس حافظه 0x7c00 بارگذاری کرده و اجرای آن را آغاز می‌کند . راهنمایی سوال هم اشاره می‌کند که با بررسی خروجی دستور make -n می‌توان فهمید محتوای کدام فایل در سکتور بوت قرار می‌گیرد.

(13

فایل باینری مربوط به بوت لودر (که در سکتور اول دیسک قرار می‌گیرد) از نوع **باینری خام** (raw binary) است . دلیل استفاده از این نوع فایل این است که BIOS کامپیوتر فقط محتوای ۵۱۲ بایتی سکتور اول را مستقیماً در آدرس حافظه 0x7c00 بارگذاری کرده و اجرای آن را شروع می‌کند و قابلیت تفسیر فرمت‌های پیچیده‌تر مانند ELF را ندارد . تفاوت اصلی این فایل باینری خام با سایر فایل‌های باینری xv6 (مانند هسته یا برنامه‌های سطح کاربر مثل ls) در این است که فایل‌های دیگر از فرمت ELF استفاده می‌کنند که شامل اطلاعات اضافی (متادیتا) مانند هدرها، جدول بخش‌ها و نقطه شروع برنامه برای بارگذاری صحیح است، در حالی که فایل بوت لودر فاقد این اطلاعات است . برای تبدیل این فایل باینری خام به زبان اسمبلی قابل خواندن، می‌توان از ابزار objdump استفاده کرد. به عنوان مثال، ممکن است دستوری شبیه به objdump -D -b binary -m i8086 bootblock (با فرض اینکه فایل باینری bootblock نام دارد) لازم باشد تا کد ماشین ۱۶ بیتی آن دیس‌اسمبل شود؛ خروجی این دستور باید با محتوای فایل bootasm.S مطابقت داشته باشد.

(14

objcopy برای حذف اطلاعات اضافی (مانند هدر ELF و متادیتا های اشکال زدایی) و تبدیل فایل ها به فرمت بانری خام استفاده می شود. این کار باعث می شود که فایل ها تولید شده مستقیما توسط بوت لودر یا کرنل در حافظه بارگذاری و اجرا شوند.

(15

فرایند بوت سیستم‌عامل xv6 با استفاده از هر دو فایل bootasm.S (نوشته شده به زبان اسمبلی) و bootmain.c (نوشته شده به زبان C) انجام می‌شود، زیرا استفاده تنها از کد C برای کل فرآیند بوت امکان‌پذیر نیست. دلیل اصلی این است که پردازنده x86 در ابتدای روشن شدن در مد حقیقی ۱۶ بیتی (real mode) شروع به کار می‌کند. در این حالت، برای انجام عملیات اولیه‌ی ضروری مانند غیرفعال کردن وقفه‌ها (دستور cli)، تنظیم اولیه ثبات‌های قطعه (segment registers)، فعال‌سازی خط A20 برای دسترسی به حافظه بیشتر از یک مگابایت، بارگذاری جدول توصیفگر عمومی (GDT)، و مهم‌تر از همه، تغییر حالت پردازنده به مد حفاظت‌شده ۳۲ بیتی (protected mode)، نیاز به استفاده از دستورالعمل‌های خاص زبان اسمبلی است که مستقیماً با سخت‌افزار و رجیسترهای کنترلی پردازنده کار می‌کنند . زبان C استاندارد، ابزارهای مستقیمی برای انجام این سطح از عملیات کنترلی پردازنده و تغییر مد آن ارائه نمی‌دهد. بنابراین، فایل bootasm.S وظیفه انجام این تنظیمات اولیه و تغییر مد را بر عهده دارد و پس از قرار دادن پردازنده در حالت ۳۲ بیتی و تنظیم پشته اولیه، کنترل را به تابع bootmain در فایل bootmain.c می‌سپارد تا بقیه مراحل بوت، مانند بارگذاری هسته از دیسک، با استفاده از زبان C انجام شود.

(16

در معماری x86، نمونه‌هایی از انواع رجیسترها و وظایف مختصر آن‌ها به شرح زیر است: یک **رجیستر عام منظوره** مانند **eax** (Extended Accumulator Register)) وجود دارد که عمدتاً برای انجام محاسبات ریاضی، نگهداری موقت داده‌ها و نگهداری مقدار بازگشتی توابع استفاده می‌شود. یک **رجیستر قطعه** مانند **cs** (Code Segment Register) وجود دارد که در مد حفاظت‌شده، به عنوان یک انتخابگر (Selector) به توصیفگر (Descriptor) قطعه‌ای از حافظه اشاره می‌کند که حاوی دستورالعمل‌های در حال اجرا است . **eflags** نمونه‌ای از یک **رجیستر وضعیت** (یا فلگ‌ها) است که بیت‌های مختلف آن وضعیت پردازنده یا نتیجه‌ی آخرین عملیات انجام شده (مانند بیت صفر یا سرریز) و همچنین فلگ‌های کنترلی (مانند فلگ فعال بودن وقفه‌ها - IF) را نگهداری می‌کند. در نهایت، یک **رجیستر کنترلی** مانند **cr0** وجود دارد که حاوی بیت‌هایی برای کنترل حالت کلی پردازنده است، از جمله بیت فعال‌سازی مد حفاظت‌شده (PE) و بیت فعال‌سازی صفحه‌بندی حافظه (PG).

(17

رجیستر عام منظوره مانند **eax** (مخفف Extended Accumulator Register): این رجیستر عمدتاً برای انجام محاسبات ریاضی، نگهداری موقت داده‌ها و همچنین برای نگهداری مقدار بازگشتی از توابع استفاده می‌شود. در سیستم‌عامل xv6، از این رجیستر برای نگهداری شماره فراخوانی سیستمی نیز استفاده می شود.

رجیستر قطعه مانند **cs** (مخفف Code Segment Register) : در مد حفاظت‌شده، این رجیستر به عنوان یک انتخابگر (Selector) عمل کرده و به توصیف گری (Descriptor) در جدول GDT اشاره می‌کند که آن توصیفگر، قطعه‌ای از حافظه را مشخص می‌کند که حاوی دستورالعمل‌های در حال اجرای برنامه است.

eflags نمونه‌ای از یک **رجیستر وضعیت** (یا فلگ‌ها) : بیت‌های مختلف این رجیستر، وضعیت فعلی پردازنده یا نتیجه‌ی آخرین عملیات انجام شده (مانند فلگ صفر یا سرریز) را نشان می‌دهند. علاوه بر این، حاوی فلگ‌های کنترلی مهمی مانند فلگ فعال بودن وقفه‌ها (IF) است.

رجیستر کنترلی مانند **cr0** : این رجیستر حاوی بیت‌هایی برای کنترل حالت کلی پردازنده است. برای مثال، بیت PE (Protection Enable) برای فعال‌سازی مد حفاظت‌شده و بیت PG (Paging) برای فعال‌سازی مکانیزم صفحه‌بندی حافظه در این رجیستر قرار دارند.

(18

اصلی‌ترین نقص مد حقیقی (real mode) در پردازنده‌های x86، محدودیت شدید آن در آدرس‌دهی حافظه است. در این مد، دسترسی به حافظه به حداکثر **۱ مگابایت** محدود می‌شود. این حجم از حافظه برای اجرای سیستم‌عامل‌های امروزی کاملاً ناکافی است و به همین دلیل، یکی از اولین اقدامات بوت‌لودر، تغییر حالت پردازنده به مد حفاظت‌شده (protected mode) است. در مورد اینکه آیا پردازنده‌های دیگر مانند ARM یا RISC-V نیز دارای مدهای مشابهی هستند یا خیر، در مستندات ارائه شده اطلاعاتی وجود ندارد.

(19

۱. به عنوان قراردادی میان بسیاری از سیستم های عامل آنها را در این آدرس load می کنند
۲. رزرو بودن آدرس های پایین برای BIOS و Bootloader .حافظه زیر 1MB معمولا توسط BIOS، کارت گرافیک و سخت افزارها اشغال می شود و آدرس 100000x0 اولین بخش خالی و ایمن برای بارگذاری کرنل است
۳. وجود حافظه کافی برای load کردن kernel که بتواند عملیات های مورد نظرش را به راحتی اجرا کند.

قابلیت های کاربردی کنسول

1) کاربر با استفاده از این دو اینتراپت Arrow up & down می تواند بین دستورات اخیری که وارد کرده جابه جا شود.

(2

پیاده سازی clear console به این صورت که به شکل یک برنامه سطح کاربر نوشته شود که کنسول را clear کند و کرسر را به سر خط برگرداند همچنین با اینتراپت + ctrl L این برنامه سطح کاربر باید از طریق shell فراخوانی و اجرا شود

(3

با یک اینتراپت مثل ctrl + H باید لاگ دستوراتی را نشان بدهد که کاربر قبلا استفاده کرده و جزو برنامه های سطح کاربر باشند

اشکال زدایی

(20

برای مشاهده breakpoint ها می توان از دستور info break یا info breakpoints استفاده کرد

```
(gdb) break cat.c:8
Breakpoint 1 at 0xf0: file cat.c, line 8.
(gdb) break cat.c:20
Breakpoint 2 at 0xfe: file cat.c, line 20.
(gdb) break cat.c:27
Breakpoint 3 at 0x0: file cat.c, line 27.
(gdb) break cat.c:40
Breakpoint 4 at 0x54: file cat.c, line 40.
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x000000f0 in cat at cat.c:8
2       breakpoint     keep y   0x000000fe in cat at cat.c:20
3       breakpoint     keep y   0x00000000 in main at cat.c:27
4       breakpoint     keep y   0x00000054 in main at cat.c:40
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x000000f0 in cat at cat.c:8
2       breakpoint     keep y   0x000000fe in cat at cat.c:20
3       breakpoint     keep y   0x00000000 in main at cat.c:27
4       breakpoint     keep y   0x00000054 in main at cat.c:40
```

(21

برای حذف breakpoint ها می توان از دو دستور زیر استفاده کرد:
d i که i شماره breakpoint ای است که باید حذف شود
delete i که i شماره breakpoint ای است که باید حذف شود

```
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x000000f0 in cat at cat.c:8
2       breakpoint     keep y   0x000000fe in cat at cat.c:20
3       breakpoint     keep y   0x00000000 in main at cat.c:27
4       breakpoint     keep y   0x00000054 in main at cat.c:40
(gdb) d 2
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x000000f0 in cat at cat.c:8
3       breakpoint     keep y   0x00000000 in main at cat.c:27
4       breakpoint     keep y   0x00000054 in main at cat.c:40
(gdb) delete 4
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x000000f0 in cat at cat.c:8
3       breakpoint     keep y   0x00000000 in main at cat.c:27
```

(22

درواقع bt مخفف backtrace می باشد. این دستور نشان دهنده call stack تا نقطه ای می باشد که برنامه متوقف شده. خطوطی که در خروجی نیز نمایش داده شده اند در واقع توابع صدا شده اند که در بالا ترین خط تابعی است که برنامه در آن متوقف شده.

```
(gdb) break cat.c:8
Breakpoint 1 at 0xf0: file cat.c, line 8.
(gdb) break cat.c:18
Breakpoint 2 at 0xd3: file cat.c, line 18.
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 2, cat (fd=3) at cat.c:18
18          if(n < 0){
(gdb) bt
#0  cat (fd=3) at cat.c:18
#1  0x00000054 in main (argc=2, argv=0x2fe4) at cat.c:39
```

در این مثال تابع بالایی یعنی cat تابعی است که برنامه متوقف شده و با #0 نشان داده شده تابعی که cat را صدا زده main است که با #1 نمایش داده می شود

(23

از دستور x برای مشاهده یک آدرس در حافظه استفاده می شود:

Syntax: x / format address

از دستور print برای نمایش یک variable یا expression استفاده می شود:

Syntax:print expression یا print variable

برای مشاهده مقدار یک رجیستر خاص می توان از دستور info registers registername استفاده کرد

(24

برای مشاهده وضعیت رجیستر ها می توان از دستور info registers استفاده کرد

```
(gdb) info registers
eax            0x0                 0
ecx            0x0                 0
edx            0x663               1635
ebx            0x0                 0
esp            0x0                 0x0 <main>
ebp            0x0                 0x0 <main>
esi            0x0                 0
edi            0x0                 0
eip            0xfff0              0xfff0
eflags         0x2                 [ IOPL=0 ]
cs             0xf000              61440
ss             0x0                 0
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
fs_base        0x0                 0
gs_base        0x0                 0
k_gs_base      0x0                 0
cr0            0x60000010          [ CD NW ET ]
cr2            0x0                 0
cr3            0x0                 [ PDBR=0 PCID=0 ]
cr4            0x0                 [ ]
--Type <RET> for more, q to quit, c to continue without paging--c
cr8            0x0                 0
efer           0x0                 [ ]
xmm0           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0,
 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm1           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0,
 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm2           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0,
 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm3           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0,
 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm4           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0,
 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm5           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0,
 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm6           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0,
 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm7           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0,
 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
mxcsr          0x1f80              [ IM DM ZM OM UM PM ]
```

برای متغیر های محلی نیز از دستور info locals استفاده می شود

```
(gdb) break cat.c:8
Breakpoint 1 at 0xf0: file cat.c, line 8.
(gdb) break cat.c:18
Breakpoint 2 at 0xd3: file cat.c, line 18.
```

```
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 2, cat (fd=3) at cat.c:18
18          if(n < 0){
(gdb) info locals
n = 0
(gdb)
```

edi(extended destination index):

ین رجیستر عنوان pointer به مقصد در اعمال string عمل می کند.در برخی از عملیات ها مانند MOVS و LODS رجیستر edi به buffer مقصد اشاره می کند

esi(extended source index):

به عنوان رجیستر مبدا در اعمال string استفاده می شود.این رجیستر معمولا به buffer مبدا اشاره می کند که داده ها خوانده می شوند

(25

input struct دارای یک بافر (input.buf) و 3 نشانگر بر روی آخرین مقدار وارد شده به بافر هستند. از آنجایی که تمامی ورودی ها داخل بافر مانند یک log ذخیره می شوند (اگر از ماکزیمم حافظه input.buf بیشتر باشد overwrite می شود) برای دسترسی به آخرین مقدار وارد شده درون این بافر نیازمند این 3 نشانگر هستیم.

مثال : system_

Input.r : به اولین کاراکتر وارد شده در کنسول اشاره دارد (در این مثال s)

Input.w : به آخرین کاراکتر وارد در کنسول اشاره دارد (در این مثال m)

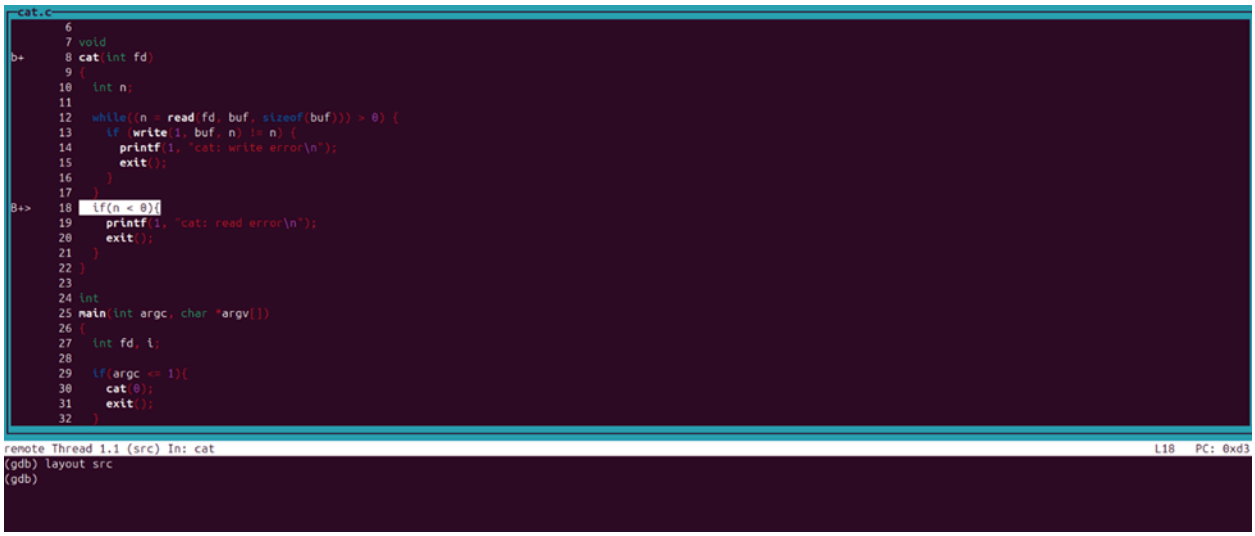
Input.c : به عنوان ایندکس ادیتور است، یعنی کاراکتر بعدی که در کنسول وارد شود داخل این ایندکس ذخیره خواهد شد (_)

```
(gdb) ptype input
type = struct {
    char buf[128];
    uint r;
    uint w;
    uint e;
}
```

همانطور که در بخش قبل گفته شد input.buf همواره در ایندکس input.e ورودی می گیرد. تنها نکته ای که باید مورد توجه واقع شود lock کرد کنسول هنگام وارد کردن یک ورودیست. به این دلیل که اگر کنسول در زمان اینتراپت با یک پردازنده، از طرف همان پردازنده لاک نشود. احتمال تغییر بافر هنگام پردازش وجود دارد. 3 نشانگر دیگر نیز همواره با تغییر ورودی کنسول به شکل گفته شده خواهند کرد.

(26

خروجی دستور src layout:



خروجی دستور asm layout:

(27

از دو دستور می توان استفاده کرد. یکی دستور up است که از تابع کنونی خارج و وارد تابعی می شود که تابع کنونی را فراخوانی کرده. دستور down از تابع کنونی خارج شده و وارد تابع بعدی که این تابع آن را فراخوانی می کند می شود