

گزارش پروژه اول آزمایشگاه درس سیستم عامل

پریا پاسبورز 810101393

کوثر شیری جعفرزاده 810101456

پریسا یحیی پور پورفتیده 810101551

لینک repository گیتهاب پروژه:

<https://github.com/parisa-yahyapour/Operating-System-Lab-01-F03/tree/main/Lab01>

هش آخرین commit:

ab572f65af8e29c06dce14c0490eb57d37232135

مقدمه‌ای درباره سیستم عامل و xv6

1. سه وظیفه اصلی سیستم عامل را نام ببرید.

1- مدیریت منابع (این مدیریت می تواند شامل منابع physical یا logical باشد مثل memory management, filesystem management and process management)

2- ایجاد نوعی واسطه بودن/مصالحه بین لایه بالایی (user application and programs) و لایه ی پایینی (سخت افزار دستگاه)

3- مدیریت کاربران و برنامه های کاربردی

به نحوی دیگر می توان گفت :

- 1- Resource management (the hardware and the software resources of the program, it insures that resources are allocated efficiently and fair among users and processes)
- 2- User interface (such as command line or GUI)
- 3- System security and Access control

2. فایل های اصلی سیستم عامل xv6:

1. Basic Headers

-دیتا تایپ هایی که در kernel استفاده می شود.

-پارامتر و constant ها نظیر محدودیت های زمانی برای یک process و همچنین constant که در kernel مورد استفاده قرار می گیرد

-شرح ساختار مموری برای kernel و همچنین فضای مورد نیاز کاربر

-ساختاری برای هندل کردن executable and linkable format binary files

-در برداشتن ساختارهای زمانی و همچنین توضیح سخت افزاری xv6

2. System Calls

- این بخش یک رابط بین user program و kernel می باشد.

- نحوه ی پیاده سازی system call ها (که شامل پردازش درخواست ورودی و هدایت آن ها به بخشی که به طور مناسب به درخواست رسیدگی کند)

3. Processes

-این بخش به مدیریت process ها می پردازد که شامل نحوه ایجاد آن ها، زمانبندی (scheduling) و نحوه پایان آن ها می شود.

-این بخش توابع و ساختارهایی را در اختیار دارد که برای multitasking ضروری هستند.

4. File System

- این بخش مسئول مدیریت فایل ها و directories می باشد

-هسته ی اصلی عملیات های file system نظیر نوشتن، خواندن، ساماندهی اطلاعات بر روی فضاهای ذخیره سازی در این بخش است.

5. User-Level Programs

-این بخش شامل برنامه های کاربر می شود که در لایه ی بالایی سیستم عامل اجرا می شود همانند shell

-این بخش مسئول فراهم آوردن user interface برای تعامل کاربر با kernel می باشد

6. Entering xv6

-این بخش برای initial setup and bootstrapping در سیستم عامل می باشد که مطمئن می شود همه چیز برای اجرای برنامه کاربر آماده است

7. Bootloader

-این بخش شامل فایل هایی می شود که وظیفه دارند در هنگام boot شدن، فایل kernel OS را در مموری لود کنند و شرایط اولیه برای اجرای سیستم عامل را فراهم کنند.

8. Linker

-این بخش شامل فایلی می شود که پارت های مختلف kernel را باهم ادغام کرده و آن را به یک فایل باینری قابل اجرا تبدیل می کند.

9. Locks:

-این بخش برای مدیریت دسترسی process ها به منابع مشترک است، اگر سیستم به صورت multiprocessing عمل کند دسترسی هر یک از آن ها به منابع و حفظ یکپارچگی داده ها اهمیت پیدا می کند.

10.Pipes:

این بخش برای ارتباط بین process ها و اجازه دادن به آن ها برای فرستادن و دریافت دیتا می باشد.

11.Low-Level Hardware:

-این بخش مدیریت تعاملات با HW نظیر ورودی و خروجی ها و ایجاد توابع برای تعامل با drivers -رسیدگی به hardware interrupt

12.String Operations:

-این بخش شامل اعمالی می شود که تغییراتی را بر روی string پیاده سازی می کند.

موارد ذکر شده از قبیل file system, kernel core, header files همگی در Linux kernel source tree قابل دسترس هستند که در ادرس زیر قرار دارد:

`/usr/src/linux`

محتویات این فایل عبارت است از:

1. Core of the OS (Kernel)

Directory: kernel/

این بخش شامل اعمالی از جمله process management, scheduling, system calls, and memory management می شود.

2. Header Files

Directory: include/

مشخصات ساختارهای داده، constants و function prototypes برای توسعه kernel می شود. می تواند شامل معماری های خاص برای معماری های متفاوت cpu نیز باشد.

3. File System

Directory: fs/

توابعی که نحوه ی دسترسی و تغییر فایل ها از جمله باز کردن، خواندن، نوشتن و بستن آن ها را تعریف می کند.

داده ساختارهای ضروری برای file management را نیز تهیه می کند.

کامپایل سیستم عامل xv6

3. دستور **make -n** را اجرا کنید کدام دستور فایل نهایی هسته را می سازد؟

دستوری که هسته را می سازد برابر دستور زیر است:

```
ld -m elf_i386 -T kernel.ld -o kernel entry.o bio.o console.o exec.o file.o fs.o ide.o ioapic.o kalloc.o kbd.o lapic.o log.o main.o mp.o picirq.o pipe.o proc.o sleeplock.o spinlock.o string.o swtch.o syscall.o sysfile.o sysproc.o trap.o asm.o trap.o uart.o vectors.o vm.o -b binary initcode entryother
```

این دستور تمامی object ها را به یکدیگر لینک می کند تا فایل هسته ی سیستم عامل به نام kernel را ایجاد کند.

4. در **Makefile** متغیرهایی به نام **UPROGS** و **ULIB** تعریف شده است. کاربرد آن ها چیست؟

1- UPROGS: User Programs

این متغیر شامل لیستی از برنامه های کاربر است که همراه با کامپایل 6xv ، برنامه های ذکر شده در این لیست نیز کامپایل شده و در دسته فایل های قابل اجرا توسط سیستم عامل قرار می گیرند.

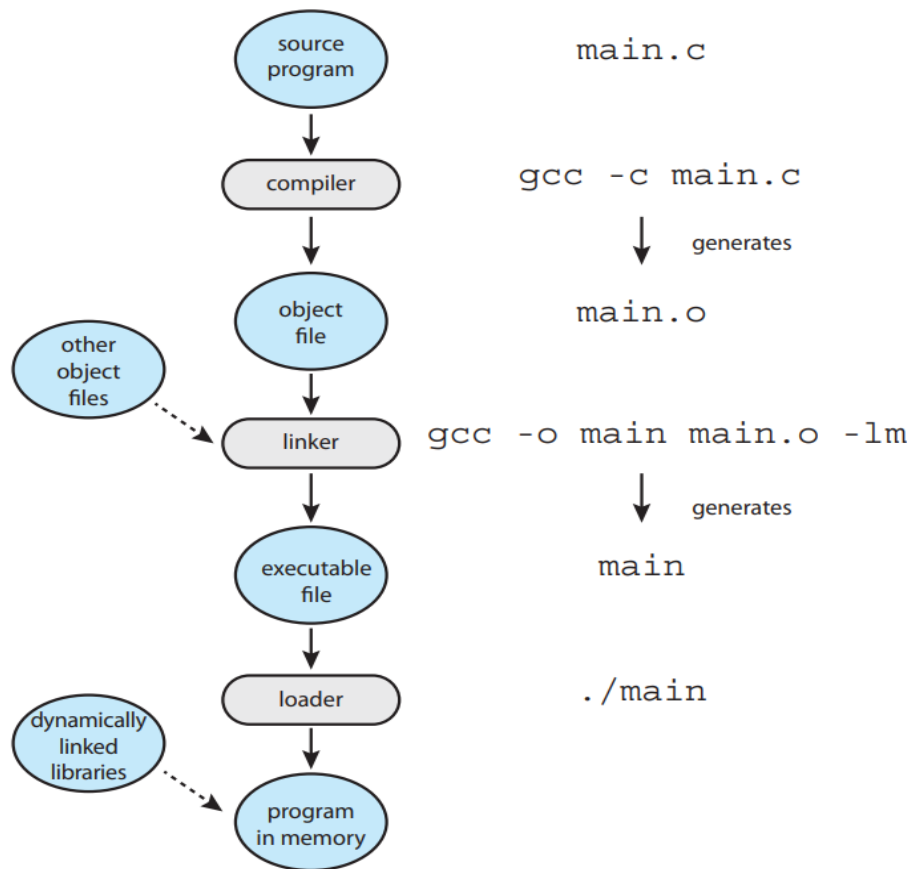
از نمونه برنامه هایی که در این بخش قرار می گیرند می توان به (`_init`, `_rm`, `ls`) اشاره کرد که همگی به فرم `filename_` هستند.

2-ULIB: User Libraries

در کدهای 6xv از اعدادی از کتابخانه های C استفاده شده است بنابراین نیاز داریم تا برای اجرای آن برنامه ها این کتابخانه ها را نیز کامپایل کنیم.

لازم است ذکر کنیم که برنامه های سطح کاربر نیازمند فایل ULIB هستند که اجرا شود.

این کتابخانه ها در ادامه توسط دستور ld به فایل های قابل اجرا اضافه می شوند.



اجرا بر روی شبیه‌ساز QEMU

5. دستور `make qemu -n` را اجرا نمایید. دو دیسک به عنوان ورودی به شبیه ساز داده شده است محتوای آن ها چیست :

```
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
```

دو دیسک به شبیه ساز داده می شود که به نام های :

1-fs.img

این دیسک در حقیقت file system مربوط به سیستم عامل xv6 را دارد که شامل user programs and necessary files می شود.

2-xv6.img

این دیسک شامل kernel core operating system به همراه bootloader برای بوت کردن سیستم عامل است.

مراحل boot سیستم عامل xv6

8. علت استفاده از دستور **objcopy** در حین عملیات **make** چیست؟

این دستور محتویات یک فایل object ورودی را در یک فایل object خروجی کپی می کند لازم به ذکر است که با توجه به استفاده این دستور از کتابخانه BFD برای ترجمه فایل مبدا به مقصد، نیازی نیست که هردو فرمت یکسانی داشته باشند چرا که این کتبخانه تمامی فرمت ها را پشتیبانی می کند.

این دستور به منظور انجام عملیات کپی از فایل های temporary کمک می گیرد و در نهایت با به اتمام رسیدن کار ترجمه انتقال آن هارا پاک می کند.

علت استفاده در make مربوط به 6xv

1- محتویات بخش txt فایل bootblock.o را در یک فایل raw binary به نام bootblock کپی می کند.

```
objcopy -S -O binary -j .text bootblock.o bootblock  
./sign.pl bootblock
```

2- در این بخش محتویات txt فایل bootblockother.o را در یک فایل raw binary به نام entryother کپی می کند

```
objcopy -S -O binary -j .text bootblockother.o entryother
```

3- در اخر نیز محتویات initcode.out داخل یک فایل raw binary به نام initcode ذخیره می شود.

```
objcopy -S -O binary initcode.out initcode
```

این فایل های باینری قبل تر در هنگام لینک کردن kernel استفاده شده بودند.

```
ld -m elf_i386 -T kernel.ld -o kernel entry.o bio.o console.o exec.o file.o f  
s.o ide.o ioapic.o kalloc.o kbd.o lapic.o log.o main.o mp.o picirq.o pipe.o proc  
.o sleeplock.o spinlock.o string.o switch.o syscall.o sysfile.o sysproc.o trapasm  
.o trap.o uart.o vectors.o vm.o -b binary initcode entryother  
objdump -S kernel > kernel.asm
```

13. کد bootmain.c هسته را با شروع از سکتور بعد از سکتور بوت خوانده و در آدرس 0x100000 قرار می دهد. علت انتخاب این آدرس چیست؟

قرار دادن هسته در آدرس 0x100000 قرار دادن هسته در آدرس 0x100000 یعنی 1 مگابایت زیر این حافظه قرار دارد. می دانیم وجه اشتراک تمام سیستم ها حافظه پایین است. یعنی حد بالای حافظه در سیستم ها متفاوت است اما خانه های پایینی حافظه به طور قراردادی برای اجزای سیستم در نظر گرفته می شود به عنوان مثال برای BIOS یا bootloader.

بنابراین برای جلوگیری از overwrite شدن این بخش مهم از سیستم، هسته xv6 در آدرس 1000000x قرار می‌گیرد.

اجرای هسته xv6

18. علاوه بر صفحه‌بندی در حد ابتدایی از قطعه‌بندی به منظور حفاظت از هسته استفاده خواهد شد. این عملیات توسط `seginit()` انجام می‌گیرد. همان‌طور که ذکر شد، ت قرار می‌گیرد. ترجمه قطعا تاثیری بر ترجمه آدرس منطقی نمی‌گذارد. زیرا تمامی قطعه‌ها اعم از کد و داده روی هم می‌افتند. با این حال برای کد و داده‌های سطح کاربر پرچم `SEG_USER` تنظیم شده‌است. چرا؟

1. Privilege level enforcement & CPU Protection Mechanism

دقت داشته‌باشید که ترجمه تاثیری روی ترجمه آدرس منطقی نمی‌گذارد، ولی همچنان CPU باید از privilege level هنگام دسترسی به حافظه مطلع باشد. با استفاده از پرچم `SEG_USER` می‌توانیم privilege level را برای یک segment در حالت ring 3 قرار دهیم که در واقع همان سطح دسترسی کاربران است.

بدون استفاده از این پرچم، Process هایی که در user mode اجرا می‌شوند به خوبی از kernel mode ایزوله نمی‌شوند و می‌توانند به نقاطی از حافظه دسترسی داشته‌باشند که مشکلات security در پی خواهد داشت.

2. System Call Transitions

وقتی یک user process یک system call ایجاد می‌کند، به تبع آن احتیاج است که از user mode به kernel mode برویم. پس CPU نیز می‌بایستی از ring 3(user mode) به ring 1(kernel mode) منتقل شود. پرچم `SEG_USER` به CPU کمک می‌کند این جابه‌جایی را انجام دهد و مطمئن باشیم که process های مربوط user در kernel mode اجرا نمی‌شوند.

در حقیقت وظیفه paging مشخص کردن این است که چه page هایی در دسترس هستند، در حالی که پرچم `SEG_USER` مشخص می‌کند چه کسی به آن دسترسی دارد.

معادل `proc struct` در لینوکس، `/proc filesystem` است.

اجرای نخستین برنامه سطح کاربر

19. جهت نگه‌داری اطلاعات مدیریتی برنامه‌های سطح کاربر ساختاری تحت عنوان `struct proc` ارائه شده‌است. اجزای آن را توضیح داده و ساختار معادل آن را در سیستم عامل لینوکس بیابید.

معماری xv6 اجزای هر process را در ساختاری تحت عنوان `struct proc` گرد هم می‌آورد. مهم‌ترین اجزای یک process در kernel state شامل `page table`، `Kernel stack` و `run state` می‌شوند.

1. Kernel stack

هر process یک thread of execution دارد که دستورات آن process را اجرا می‌کند. یک thread می‌تواند غیرفعال و مجدداً فعال شود. برای جابه‌جایی بین process‌های مختلف، kernel، thread مربوط به یک process را غیرفعال کرده و thread مربوط به process دیگری را فعال می‌کند. بیشتر وضعیت یک thread که شامل متغیرهای محلی، function call و آدرس بازگشت می‌شود در kernel stack ذخیره می‌شود.

وقتی یک process دستورات مربوط به user را اجرا می‌کند، فقط user stack استفاده می‌شود و kernel stack خالی است. از طرفی وقتی سیستم بخاطر یک system call یا interrupt وارد kernel mode می‌شود، kernel stack استفاده می‌شود ولی user stack اطلاعات موجود در خودش را حفظ می‌کند، ولی به طور فعال مورد استفاده نیست.

یک process مرتباً بین استفاده از kernel stack و user stack جابجا می‌شود. البته kernel stack از user stack جدا و حفاظت شده است و اگر process ای user stack را دچار اشکال کند، Kernel می‌تواند اجرا شود.

یک thread می‌تواند kernel را متوقف کند و برای انجام عملیات IO صبر کند و پس از اتمام کار IO کارش را از جایی که متوقف شده بود از سر بگیرد.

2. Run state

این جزء نشان می‌دهد که یک process در حال حاضر:

- i. حافظه‌ای برای آن اختصاص یافته است.
- ii. آماده برای اجراست.
- iii. در حال اجراست.
- iv. منتظر عملیات IO است.
- v. در حال اتمام است (exiting).

3. Page table

در این بخش page table مربوط به process به شکلی که مورد انتظار معماری xv6 است ذخیره می‌شود. Paging hardware در هنگام اجرای یک Process از Page table استفاده می‌کند. همچنین از آن می‌توان برای نگهداری آدرس‌های فیزیکی‌ای استفاده کرد که برای حفظ حافظه process اختصاص یافته شده‌اند.

23. کدام بخش از آماده‌سازی سیستم بین تمام هسته‌های پردازنده مشترک و کدام بخش اختصاصی است؟ زمان‌بند روی کدام هسته اجرا می‌شود؟

در معماری xv6، تعداد core‌ها به تنظیمات سیستمی که روی آن شبیه‌سازی یا اجرا می‌شود، بستگی دارد. اما وقتی روی سیستم‌های شبیه‌سازی مثل qemu اجرا شود، از دو هسته پشتیبانی می‌کند.

مسئولیت‌های مشترک هسته‌های پردازنده عبارتند از:

1. Process scheduling

همه هسته‌ها می‌توانند اجرای process‌ها را با کمک صف مشترک بین خودشان زمان‌بندی کنند. برای ایجاد تعادل بین حجم کار درون سیستم، هر هسته به محض بیکار شدن انجام یک process را برعهده می‌گیرد.

2. Inter-core communication:

هسته‌ها با استفاده از یک حافظه مشترک یا مکانیزم‌های خاصی مثل IPI با یکدیگر در ارتباط هستند تا کارها را مدیریت کنند.

3. Memory Management

معماری xv6 یک حافظه مشترک دارد که تمام هسته‌ها به آن دسترسی دارند. آنها باید مسائلی نظیر Load، page fault کردن صفحه‌ها داخل حافظه و ترجمه آدرس به کمک page table را انجام دهند.

4. Kernel Mode and System Calls

هر هسته می‌تواند به kernel mode تغییر وضعیت دهد تا system call ای که از طرف یک process سطح کاربر ایجاد شده را مدیریت کند. توانایی هندل کردن این system call‌ها بین هسته‌ها مشترک است.

5. Locking and Synchronization

از آنجایی که هر هسته می‌تواند به منابع مشترک اعم از حافظه، process table و device‌ها دسترسی داشته‌باشد، معماری xv6 از قفل‌ها استفاده می‌کند تا مطمئن باشد که چندین operation به طور همزمان صورت نمی‌گیرد. همه هسته‌ها باید locking mechanism را رعایت کنند.

مسئولیت‌های خاص هسته‌های پردازنده عبارتند از:

1. Local Interrupt handling

هر هسته باید interrupt‌های سخت‌افزاری‌اش را به تنهایی مدیریت کند.

2. Context Switching

هر هسته اطلاعات مربوط به process ای که اجرا می کند، اعم از register ها و stack pointer ها را درون خودش نگه می دارد. پس context switching هم باید توسط هر هسته به تنهایی انجام شود، چون فقط او از اطلاعات process آگاه است.

3. Core initialization

وقتی سیستم boot می شود، bootstrap processor اولین هسته ای است که شروع به کار می کند. این هسته environment اولیه را ست می کند و سپس application processor شروع به کار می کنند (بقیه هسته ها).

همانطور که توضیح داده شد، زمان بندی یک فعالیت مشترک بین هسته های سیستم است و به هسته خاصی محدود نمی شود.

اشکال زدایی

1. برای مشاهده breakpoints از چه دستوری استفاده می شود؟

برای این منظور از دستور info breakpoints استفاده می شود که تمامی break point ها را نمایش می دهد:

```
(gdb) break cat.c:15
Breakpoint 1 at 0xeb: file cat.c, line 15.
(gdb) break cat.c:27
Breakpoint 2 at 0x4: file cat.c, line 27.
(gdb) break cat.c:34
Breakpoint 3 at 0x5c: file cat.c, line 34.
(gdb) info breakpoints
Num      Type             Disp Enb Address          What
1        breakpoint      keep y   0x000000eb in cat at cat.c:15
2        breakpoint      keep y   0x00000004 in main at cat.c:27
3        breakpoint      keep y   0x0000005c in main at cat.c:34
(gdb)
```

همانطور که مشخص است هر 3 break point قرار گرفته را نشان می دهد.

2. برای حذف breakpoints از چه دستوری استفاده می کنیم؟

برای حذف آن ها از دستور breakpoint number "del" استفاده می کنیم که به شکل زیر عمل می کند.

ابتدا breakpoint های مد نظر را اضافه می کنیم:

```
(gdb) break cat.c:15
Breakpoint 1 at 0xeb: file cat.c, line 15.
(gdb) break cat.c:35
Breakpoint 2 at 0x30: file cat.c, line 35.
(gdb) break cat.c:10
Breakpoint 3 at 0x97: file cat.c, line 10.
(gdb) info breakpoints
Num      Type             Disp Enb Address      What
1        breakpoint      keep y   0x000000eb in cat at cat.c:15
2        breakpoint      keep y   0x00000030 in main at cat.c:35
3        breakpoint      keep y   0x00000097 in cat at cat.c:10
(gdb)
```

در قدم بعدی breakpoint دوم را حذف می زنیم. لازم به ذکر است برای انجام این عمل نیاز است که شماره آن را برای دستور وارد کنیم.

```
(gdb) del 2
(gdb) info breakpoints
Num      Type             Disp Enb Address      What
1        breakpoint      keep y   0x000000eb in cat at cat.c:15
3        breakpoint      keep y   0x00000097 in cat at cat.c:10
(gdb)
```

همانطور که مشاهده می شود breakpoint خط 35 ام حذف گردیده است.

کنترل روند اجرا و دسترسی به حالت سیستم

3. دستور bt را اجرا کنید خروجی چه چیزی را نشان می دهد؟

این دستور در هنگام توقف برنامه می تواند call stack را نشان دهد. همانطور که در تصویر مشخص است به صورت FILO نمایش داده شده است.

این دستور مخفف backtrace می باشد که هر تابع با صدا زده شدن بخشی از آن را به خود اختصاص داده است، در این بخش آدرس، متغیرهای محلی و ... دیده می شود.

```

(gdb) break cat.c:12
Breakpoint 1 at 0x97: file cat.c, line 12.
(gdb) break cat.c:34
Breakpoint 2 at 0x5c: file cat.c, line 34.
(gdb) continue
Continuing.
[Switching to Thread 1.2]

Thread 2 hit Breakpoint 1, cat (fd=0) at cat.c:12
12      while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb) bt
#0  cat (fd=0) at cat.c:12
#1  0xffffffff in ?? ()
(gdb) continue
Continuing.

Thread 2 hit Breakpoint 1, cat (fd=3) at cat.c:12
12      while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb) bt
#0  cat (fd=3) at cat.c:12
#1  0x00000054 in main (argc=<optimised out>, argv=<optimised out>) at cat.c:39
(gdb)

```

همانطور که مشخص است با رسیدن خط 39 دو عنصر در این call stack قرار دارند پارامترها و همچنین نقطه ی مد نظر نیز دیده می شود.

4. تفاوت دو دستور x و print را توضیح دهید. چگونه می توان محتوای یک ثبات خاص را مشاهده کرد؟

دستور x مخفف examine می باشد. هدف هر دو دستور بررسی مقدار داده در بخشی از حافظه می باشد اما تعدادی تفاوت هم دارند.

ابتدا دستور x را بررسی می کنیم:

هدف استفاده از این دستور، بررسی حافظه در آدرسی خاص است. در واقع این دستور این امکان را فراهم می کند که به طور مستقیم بتوان مقادیر مکان های مختلف حافظه را مشاهده کرد بدون توجه به اینکه آن خانه از حافظه به یک متغیر اختصاص داده شده یا خیر. از قابلیت های دیگر این دستور این است که می توان فرمت خروجی که می گیریم را تعیین کرد. برای مثال می توان محتوای حافظه را به شکل هگز، دسیمال یا کاراکتر مشاهده کرد. همچنین امکان تعیین واحد داده هم وجود دارد. برای مثال: بایت، کلمه و...

سینتکس این دستور به شکل زیر می باشد:

x/<n><f><u> <address>

X:دستور

<n>:تعداد واحد هایی که می خواهیم ببینیم

<f>فرمت خروجی:

<u>واحد:

<address>آدرس حافظه:

مقادیر ممکن برای پارامترها:

پارامتر f:

<f>	توضیح
x	نمایش هگز
d	نمایش دسیمال
c	نمایش به صورت کاراکتر

پارامتر u:

<f>	توضیح
b	بایت
h	نصف کلمه
w	کلمه

حال دستور print را بررسی می کنیم. هدف این دستور نشان دادن مقدار یک عبارت، متغیر، یا شیء می باشد. در واقع می توان در نظر گرفت این دستور برای دیباگ سطح بالاتری نسبت به x به کار می رود چرا که مقادیر abstract تری را به ما نشان می دهد درحالی که دستور x در پایین ترین سطح قرار دارد و با گرفتن آدرس حافظه مقدار آن را نشان می دهد فارغ از اینکه از آن حافظه به چه چیزی اختصاص دارد. سینتکس این دستور به شکل زیر است:

Print <expression/variable>

به طور کلی می توان در نظر گرفت زمانی می توانیم از x استفاده کنیم که ساختار حافظه را بدانیم و آدرس هر چیزی که می خواهیم در حافظه مشخص باشد. اما دستور print سطح بالاتر بوده و مقادیر حافظه خام را به ما نشان نمی دهد بلکه مقادیر اختصاص داده شده به متغیرها یا عبارات را نشان می دهد.

در صورتی که در gdb از دستور help استفاده کنیم توضیحات مربوط را می توان به کمک gdb مشاهده کرد.


```
(gdb) help x
Examine memory: x/FMT ADDRESS.
ADDRESS is an expression for the memory address to examine.
FMT is a repeat count followed by a format letter and a size letter.
Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal),
    t(binary), f(float), a(address), i(instruction), c(char), s(string)
    and z(hex, zero padded on the left).
Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).
The specified number of objects of the specified size are printed
according to the format.  If a negative number is specified, memory is
examined backward from the address.

Defaults for format and size letters are those previously used.
Default count is 1.  Default address is following last thing printed
with this command or "print".
(gdb) █
```

```
Print value of expression EXP.
Usage: print [[OPTION]... --] [/FMT] [EXP]

Options:
  -address [on|off]
    Set printing of addresses.

  -array [on|off]
    Set pretty formatting of arrays.

  -array-indexes [on|off]
    Set printing of array indexes.

  -elements NUMBER|unlimited
    Set limit on string chars or array elements to print.
    "unlimited" causes there to be no limit.

  -max-depth NUMBER|unlimited
    Set maximum print depth for nested structures, unions and arrays.
    When structures, unions, or arrays are nested beyond this depth then they
    will be replaced with either '{...}' or '(...)' depending on the language.
    Use "unlimited" to print the complete structure.

--Type <RET> for more, q to quit, c to continue without paging--█
```

Note: because this command accepts arbitrary expressions, if you specify any command option, you must use a double dash ("--") to mark the end of option processing. E.g.: "print -o -- myobj".

Variables accessible are those of the lexical environment of the selected stack frame, plus all those whose scope is global or an entire file.

\$NUM gets previous value number NUM. \$ and \$\$ are the last two values.

\$\$NUM refers to NUM'th value back from the last one.

Names starting with \$ refer to registers (with the values they would have if the program were to return to the stack frame now selected, restoring all registers saved by frames farther in) or else to debugger "convenience" variables (any such name not a known register).

Use assignment expressions to give values to convenience variables.

{TYPE}ADREXP refers to a datum of data type TYPE, located at address ADREXP.

@ is a binary operator for treating consecutive data objects anywhere in memory as an array. F00@NUM gives an array whose first element is F00, whose second element is stored in the space following where F00 is stored, etc. F00 must be an expression whose value resides in memory.

EXP may be preceded with /FMT, where FMT is a format letter but no count or size letter (see "x" command).

برای مشاهده مقادیر ثبات های خاص که برای آنها اسم تعریف شده برای مثال eax هم می توان از print استفاده کرد. برنامه cat_ را به کمک gdb اجرا می کنیم و در تابع main یک break point قرار می دهیم. سپس به کمک دستور زیر مقدار ثبات eax را چاپ می کنیم:

Print \$eax


```
(gdb) break cat.c:main
Breakpoint 1 at 0x0: file cat.c, line 26.
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, main (argc=2, argv=0x2fec) at cat.c:26
26      {
(gdb) print $eax
$1 = 0
```

اما برای این کار نمی توان از دستور x استفاده کرد چرا که آدرس ثابت eax را در حافظه نمی دانیم. اما اگر دستور \$eax x را اجرا کنیم، چه اتفاقی می افتد؟ دستور با مقدار درون ثابت eax مانند یک آدرس عمل کرده و محتوای آن را نشان می دهد:

```
(gdb) x $eax
0x0 <main>: 0xfble0ff3
(gdb) █
```

5. برای نمایش وضعیت ثباتها از چه دستوری استفاده می شود؟ متغیرهای محلی چگونه؟

برای مشاهده تمام رجیسترها از دستور info registers می توان استفاده کرد که اطلاعات تمام رجیسترها را به ما نشان می دهد:

```

eax      0x0      0
ecx      0x1940   6464
edx      0xbfac   49068
ebx      0xbfa8   49064
esp      0x2fe0   0x2fe0
ebp      0x3fb8   0x3fb8
esi      0x0      0
edi      0x0      0
eip      0x0      0x0 <main>
eflags   0x202    [ IOPL=0 IF ]
cs       0x1b     27
ss       0x23     35
ds       0x23     35
es       0x23     35
fs       0x0      0
gs       0x0      0
fs_base  0x0      0
gs_base  0x0      0
k_gs_base 0x0      0
cr0      0x80010011 [ PG WP ET PE ]
cr2      0x0      0
cr3      0xdfc6000 [ PDBR=0 PCID=0 ]
cr4      0x10     [ PSE ]
--Type <RET> for more, q to quit, c to continue without paging--
cr8      0x0      0

```

همچنین در صورتی که مقدار رجیستر خاصی را بخواهیم اگر آدرس آن را بدانیم از دستور x در غیر این صورت مطابق سوال 4 از دستور print استفاده می کنیم.

برای دیدن مقادیری متغیرهای محلی از دستور info locals می توان استفاده کرد که در محل رسیدن به break point مقادیر تمام متغیرهای محلی را نشان می دهد.

```

(gdb) info locals
fd = <optimized out>
i = <optimized out>

```

مقدار optimized out در حالتی رخ می دهد که کامپایلر کد را به گونه ای کامپایل کرده که نیاز به ذخیره این دو مقدار در حافظه نبوده یا مکان آن قابل شناسایی نیست. دلیل این حالت این است که کد با flagهای optimization کامپایل شده است.

رجیسترهای esi و edi از رجیسترهای general purpose هستند که معمولا برای عملیات های روی رشته کاربرد دارند. برای مثال برای مقایسه دو رشته ما یک آدرس داریم که رشته مبدا را نشان می دهد و یک آدرس داریم که رشته مقصد را نشان می دهد. رجیستر esi مخفف extended source index است که به آدرس مبدا اشاره می کند.

رجیستر edi مخفف extended destination index است که به آدرس مقصد اشاره می کند.

همچنین این دو رجیستر برای جابه جایی داده در حافظه هم کاربرد دارند.

6. به کمک استفاده از ساختار GDB، درباره ساختار struct input موارد زیر را توضیح دهید:

- توضیح کلی این ساختار و متغیرهای درونی و نقش آنها

به طور کلی از این struct برای مدیریت ورودی از سمت کنسول مورد استفاده قرار می گیرد.

ساختار کلی struct مشابه روبهرو است:

```
struct {  
    char buf[INPUT_BUF];  
    uint r; // Read index  
    uint w; // Write index  
    uint e; // Edit index  
} input;
```

buf آرایه ای است که کاراکترهای ورودی را نگه می دارد.

r ایندکسی را نشان می دهد که دفعه بعد باید از آن بخوانیم.

w ایندکسی را نشان می دهد که دفعه بعد باید در آن بنویسیم، زمانی که آماده process باشد.

e ایندکسی را نشان می دهد که باید تغییر کند.

- نحوه و زمان تغییر متغیرهای درونی

یکی از بخشهایی که از این struct استفاده می‌بریم تابع `consoleintr` است:

```
consoleintr(int (*getc)(void))
{
    int c, doprocdump = 0;

    acquire(&cons.lock);
    while((c = getc()) >= 0){
        switch(c){
            case C('P'): // Process listing.
                // procdump() locks cons.lock indirectly; invoke later
                doprocdump = 1;
                break;
            case C('U'): // Kill line.
                while(input.e != input.w &&
                    | input.buf[(input.e-1) % INPUT_BUF] != '\n'){
                    input.e--;
                    consputc(BACKSPACE);
                }
                break;
            case C('H'): case '\x7f': // Backspace
                if(input.e != input.w){
                    input.e--;
                    consputc(BACKSPACE);
                }
                break;
            default:
                if(c != 0 && input.e-input.r < INPUT_BUF){
                    c = (c == '\r') ? '\n' : c;
                    input.buf[input.e++ % INPUT_BUF] = c;
                    consputc(c);
                    if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
                        input.w = input.e;
                        wakeup(&input.r);
                    }
                }
                break;
        }
    }
    release(&cons.lock);
    if(doprocdump) {
        procdump(); // now call procdump() wo. cons.lock held
    }
}
```

✓ اولین حالت زمانی است که ترکیب `Ctrl + U` فشار داده شود:

این ترکیب منجر به پاک شدن خط کنونی کد می‌شود. برای مشخص کردن شروع خط دو حالت وجود دارد:

1. اول چک می‌شود که `buffer` خالی نباشد. همانطور که گفتیم، `input.w` جایی که کاراکتر بعدی باید نوشته شود را نشان می‌دهد و `input.e` جایی که قرار است تغییر کند. اگر این دو مخالف یکدیگر باشند، یعنی `buffer` خالی است و چیزی برای پاک کردن نداریم.

2. در این بخش چک می‌کنیم که کاراکتر آخر در `buffer` یک خط جدید نباشد. برای این کار موقعیت آخرین کاراکتر وارد شده را در نظر می‌گیریم (`input.e - 1`)، سپس باقی‌مانده‌اش به حجم `buffer` را محاسبه می‌کنیم تا ایندکس آخرین خانه پر `buffer` پیدا شود. حال با کمک ایندکس، مقدار موجود در آن خانه `buffer` را پیدا می‌کنیم.

حال مادامی که هر یک از این شرایط که مهیا بود، در یک حلقه `while` مرتباً `index.e` را یکی عقب می‌بریم و جای آن `backspace` می‌گذاریم تا زمانی که خط تمام شود.

✓ دومین حالت زمانی است که ترکیب `Ctrl + H` فشار داده شود:

از این ترکیب برای پاک کردن کاراکتر آخر استفاده می‌شود، درست قبل از جایی که `cursor` قرار دارد. مجدداً برای چک کردن اینکه `buffer` خالی نباشد، چک می‌کنیم که `input.e` و `input.w` مخالف یکدیگر باشند.

سپس `Input.e` را یکی عقب می‌بریم و به جایش `backspace` می‌گذاریم.

✓ برای هندل کردن حالت نرمال ورودی نیز از `Input` استفاده می‌کنیم.

با کمک عبارت `input.e - input.r < INPUT_BUF`، چک می‌شود که `buffer` خالی نباشد.

هر کاراکتر جدید باید در `input.buf` ذخیره شود. `Index` آن توسط `Input.e` مشخص می‌شود که یکی به جلو رفته است. چون `buffer` 128 تایی است، باید باقی‌مانده به 128 محاسبه شود.

اگر ورودی یکی از حالات خاص مثل خط جدید، پایان فایل بود یا اینکه بافر پر شده بود، باید `Input.w` را با ایندکس نقطه‌ای که در آن می‌نوشتیم آپدیت کنیم و `process` ای که روی بافر ورودی منتظر شروع به کار است را آغاز کنیم.

یکی دیگر از بخشهایی که از struct input استفاده می‌کنیم در تابع consoleread است:

```
int
consoleread(struct inode *ip, char *dst, int n)
{
    uint target;
    int c;

    iunlock(ip);
    target = n;
    acquire(&cons.lock);
    while(n > 0){
        while(input.r == input.w){
            if(myproc()->killed){
                release(&cons.lock);
                ilock(ip);
                return -1;
            }
            sleep(&input.r, &cons.lock);
        }
        c = input.buf[input.r++ % INPUT_BUF];
        if(c == C('D')){ // EOF
            if(n < target){
                // Save ^D for next time, to make sure
                // caller gets a 0-byte result.
                input.r--;
            }
            break;
        }
        *dst++ = c;
        --n;
        if(c == '\n')
            break;
    }
    release(&cons.lock);
    ilock(ip);

    return target - n;
}
```

از این تابع برای خواندن ورودی از کنسول استفاده می‌شود:

- ✓ در حلقه تودرتو، شرط برقراری حلقه دوم با کمک struct input مشخص می‌شود.
- ✓ Input.e==input.r مشخص می‌کند که ورودی تمام شده است یا خیر.
- ✓ همچنین خواندن کاراکتر جدید نیز مشابه آنچه در تابع قبل توضیح داده شد، با کمک input.buf انجام می‌شود.
- ✓ Ctrl + D سیگنالی است که نشان دهنده پایان فایل است. در صورت دریافت آن، اول باید چک کنیم که حداقل یک ورودی تاکنون از فایل خوانده باشیم ($n < \text{target}$)، اگر این طور بود input.r را یکی کم می‌کنیم تا عملیات EOF موقتاً به تاخیر بیفتد و دفعه بعد که تابع صدا زده شد، انجام شود. در غیر این صورت به انتهای فایل رسیده‌ایم و باید از تابع خارج شویم.

حال تغییرات input را در حالات مختلف با کمک gdb بررسی می کنیم:

✓ تغییرات input.e:

ابتدا متغیر input.e را watch می کنیم:

```
(gdb) watch input.e
Hardware watchpoint 1: input.e
```

اگر یک کاراکتر داخل ترمینال qemu وارد کنیم، وارد حالت دیفالت شده و مقدار Input.e در آنجا آپدیت می شود و به یک تبدیل می شود:

```
Old value = <unreadable>
New value = 1
0x801008f7 in consoleintr (getc=0x801026f0 <kbdgetc>) at console.c:218
218         c = (c == '\r') ? '\n' : c;
```

اگر u + control را وارد کنیم، برنامه در بخش مربوطه و خط 207 متوقف شده و مقدار Input.e مطابق کد یکی می شود:

```
Old value = 1
New value = 0
consoleintr (getc=0x801026f0 <kbdgetc>) at console.c:207
207         consputc(BACKSPACE);
```

سپس یک کاراکتر وارد کردیم تا مقدار Input.e یک شود و بتوانیم دستور ctrl + h را آزمایش کنیم:

```
Old value = 1
New value = 0
consoleintr (getc=0x801026f0 <kbdgetc>) at console.c:213
213         consputc(BACKSPACE);
```

✓ تغییرات input.buf :

این بار watch را روی input.buf می‌گذاریم. با هر بار نوشتن یک کاراکتر جدید، می‌توانیم آپدیت شدن مقدار input.buf را مشاهده کنیم:

```
Old value = <unreadable>
New value = "d", '\000' <repeats 126 times>
consoleintr (getc=0x801026f0 <kbdgetc>) at console.c:220
220          consputc(c);
(gdb) c
Continuing.

Thread 1 hit Hardware watchpoint 1: input.buf

Old value = "d", '\000' <repeats 126 times>
New value = "q", '\000' <repeats 126 times>
consoleintr (getc=0x801026f0 <kbdgetc>) at console.c:220
220          consputc(c);
```

✓ تغییرات input.w :

در این مرحله watch روی Input.w قرار دارد. حالتی را در نظر بگیرید که اول یک کاراکتر وارد کردیم و مقدار Input.e آپدیت شده، سپس با وارد کردن یک new line، مقدار Input.e مجدداً آپدیت شده و مقدار جدیدش درون Input.w ریخته می‌شود. این حالت با استفاده از gdb قابل مشاهده است:

```
Old value = <unreadable>
New value = "d", '\000' <repeats 126 times>
consoleintr (getc=0x801026f0 <kbdgetc>) at console.c:220
220          consputc(c);
(gdb) c
Continuing.

Thread 1 hit Hardware watchpoint 1: input.buf

Old value = "d", '\000' <repeats 126 times>
New value = "q", '\000' <repeats 126 times>
consoleintr (getc=0x801026f0 <kbdgetc>) at console.c:220
220          consputc(c);
```


✓ تغییرات input.r:

در این مرحله watch روی input.r قرار دارد. هر تعداد کاراکتر که داخل ترمینال بزنیم، به محض زدن new line، از روی کنسول خوانده می‌شود و مقدار input.r آپدیت می‌شود:

```
Old value = <unreadable>
New value = 1
0x80100321 in consoleread (ip=0x8010fa24 <icache+196>, dst=<optimized out>, n=<optimized out>) at console.c:253
253      c = input.buf[input.r++ % INPUT_BUF];
(gdb) c
Continuing.

Thread 1 hit Hardware watchpoint 1: input.r

Old value = 1
New value = 2
0x80100321 in consoleread (ip=0x8010fa24 <icache+196>, dst=<optimized out>, n=<optimized out>) at console.c:253
253      c = input.buf[input.r++ % INPUT_BUF];
(gdb) c
Continuing.

Thread 1 hit Hardware watchpoint 1: input.r

Old value = 2
New value = 3
0x80100321 in consoleread (ip=0x8010fa24 <icache+196>, dst=<optimized out>, n=<optimized out>) at console.c:253
253      c = input.buf[input.r++ % INPUT_BUF];
(gdb) c
Continuing.

Thread 1 hit Hardware watchpoint 1: input.r

Old value = 3
New value = 4
0x80100321 in consoleread (ip=0x8010fa24 <icache+196>, dst=<optimized out>, n=<optimized out>) at console.c:253
253      c = input.buf[input.r++ % INPUT_BUF];
(gdb) c
Continuing.
```

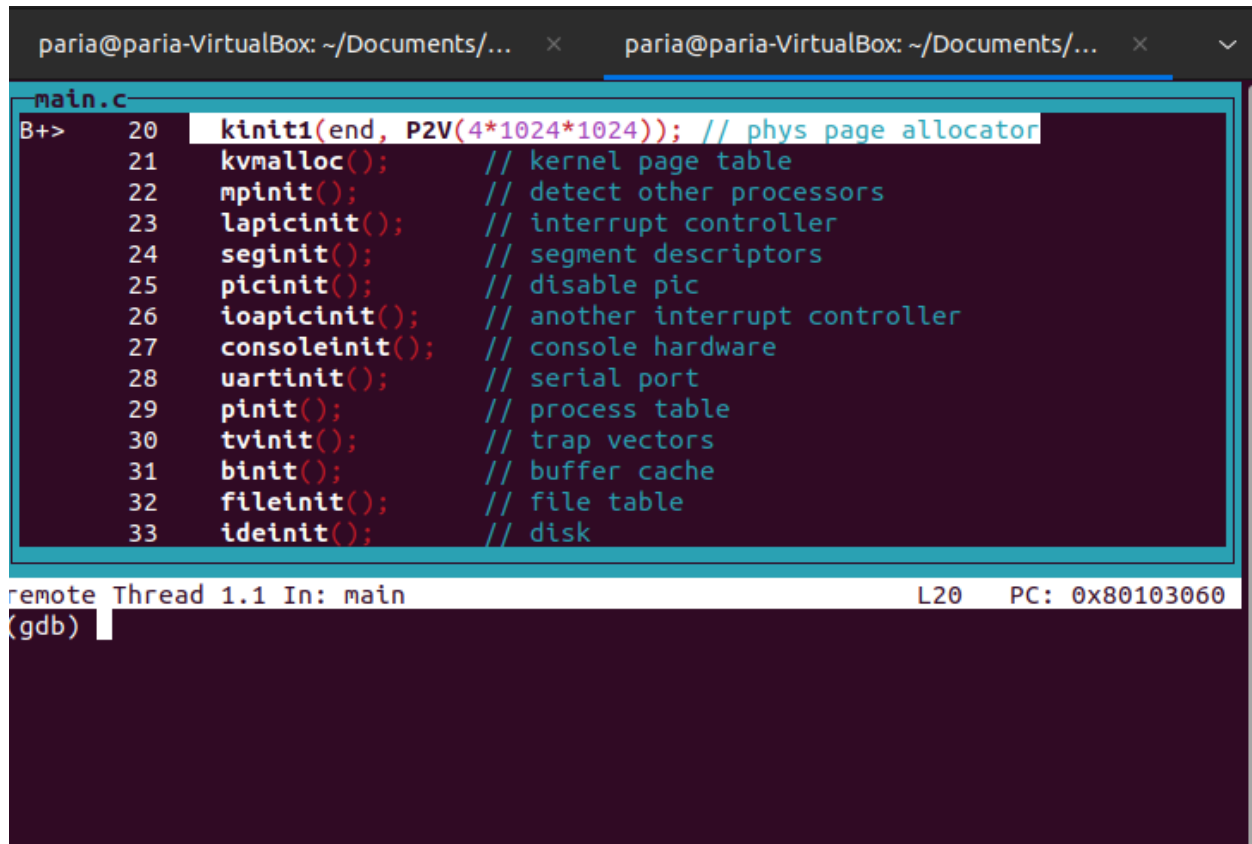
اشکال زدایی در سطح کد Assembly

7. خروجی دستوره‌های layout src و layout asm در TUI چیست؟

برای بررسی ساختار این دستورها، اول باید یک breakpoint قرار دهیم:

```
(gdb) break main
Breakpoint 1 at 0x80103060: file main.c, line 20.
```

حال دستور layout src را اجرا می‌کنیم:



```
paria@paria-VirtualBox: ~/Documents/... x paria@paria-VirtualBox: ~/Documents/... x v
main.c
B+> 20 kinit1(end, P2V(4*1024*1024)); // phys page allocator
    21 kvmalloc(); // kernel page table
    22 mpinit(); // detect other processors
    23 lapicinit(); // interrupt controller
    24 seginit(); // segment descriptors
    25 picinit(); // disable pic
    26 ioapicinit(); // another interrupt controller
    27 consoleinit(); // console hardware
    28 uartinit(); // serial port
    29 pinit(); // process table
    30 tvinit(); // trap vectors
    31 binit(); // buffer cache
    32 fileinit(); // file table
    33 ideinit(); // disk

remote Thread 1.1 In: main L20 PC: 0x80103060
(gdb)
```

این دستور source code بخشی را نشان می‌دهد که در حال debug آن هستیم.

حال دستور asm layout را اجرا می کنیم:

```
paria@paria-VirtualBox: ~/Documents/... x paria@paria-VirtualBox: ~/Documents/... x v
B+> 0x80103060 <main> lea 0x4(%esp),%ecx
0x80103064 <main+4> and $0xffffffff,%esp
0x80103067 <main+7> push -0x4(%ecx)
0x8010306a <main+10> push %ebp
0x8010306b <main+11> mov %esp,%ebp
0x8010306d <main+13> push %ebx
0x8010306e <main+14> push %ecx
0x8010306f <main+15> sub $0x8,%esp
0x80103072 <main+18> push $0x80400000
0x80103077 <main+23> push $0x801154d0
0x8010307c <main+28> call 0x80102610 <kinit1>
0x80103081 <main+33> call 0x80106ec0 <kvmalloc>
0x80103086 <main+38> call 0x80103210 <mpinit>
0x8010308b <main+43> call 0x801027f0 <lapicinit>

remote Thread 1.1 In: main L20 PC: 0x80103060
(gdb) layout asm
(gdb)
```

این دستور source code را در فرمت کد Assembly نمایش می دهد.

8. برای جابه جایی میان توابع زنجیره فراخوانی جاری (نقطه توقف) از چه دستوراتی استفاده می شود؟
از دستورات up و down برای جابه جایی میان توابع موجود در stack frame برنامه استفاده می شود.
وقتی برنامه به یک break point برخورد می کند، با این دو دستور می توان بین توابع جابه جا شد.
به عنوان مثال در دستور پایین مشاهده می کنید که یک Breakpoint روی sys_read system call گذاشته ایم و برنامه را اجرا کرده ایم تا زمانی که به یک breakpoint برخورد کنیم:

```
(gdb) break sys_read
Breakpoint 1 at 0x80104cf0: file sysfile.c, line 27.
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, sys_read () at sysfile.c:27
27         if(argint(n, &fd) < 0)
```

با کمک دستور backtrace می‌توانیم call stack مربوطه را مشاهده کنیم:

```
(gdb) backtrace
#0  sys_read () at sysfile.c:27
#1  0x80104a89 in syscall () at syscall.c:139
#2  0x80105aad in trap (tf=0x8dffefb4) at trap.c:43
#3  0x8010584f in alltraps () at trapasm.S:20
#4  0x8dffefb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
```

حال به کمک دو دستور up و down میتوان بین توابع جابه‌جا شد:

```
(gdb) up
#1  0x80104a89 in syscall () at syscall.c:139
139         curproc->tf->eax = syscalls[num]();
(gdb) down
#0  sys_read () at sysfile.c:27
27         if(argint(n, &fd) < 0)
(gdb) up
#1  0x80104a89 in syscall () at syscall.c:139
139         curproc->tf->eax = syscalls[num]();
(gdb) up
#2  0x80105aad in trap (tf=0x8dffefb4) at trap.c:43
43         syscall();
(gdb) █
```