

# آزمایشگاه سیستم عامل

آزمایش اول

استاد کارگهی

اعضا: علی پاکدل صمدی، علی کرامتی، محمد هنرجو

## آشنایی با سیستم عامل xv6

1. سیستم عامل xv6 نسخه مدرن شده از ششمین نسخه Unix می باشد که در ANSI C برای سیستم های با پروسسور x86 طراحی شده است. هسته این سیستم عامل مشابه سایر نسخه های Unix به صورت monolithic می باشد. چرا که کل سیستم عامل در حالت سوپروایز و در فضای هسته فعالیت می کند. monolithic فضایی مجازی بالاتر از بخش سخت افزاری کامپیوتر به وجود می آورد. بنابراین اگر مشکلی در سیستم عامل بوجود بیاید این باعث میشود کل سیستم با اختلال همراه شود و حتی منجر به کرش شدن سیستم عامل شود.

2. پردازنده xv6 شامل فضای کاربری حافظه (user space mem) که این بخش خودش شامل داده، پشته و دستورات است. همچنین هسته یک حالت خصوصی برای هر پردازنده فراهم می آورد. یک پردازنده با استفاده از fork یک پردازنده جدید می سازد که به آن به اصطلاح بچه پردازنده می گویند. تابع fork در نهایت بچه پردازنده به همراه پردازنده اصلی را بر میگرداند. در صورت تعدد بچه ها با استفاده از روش time sharing آن ها را مدیریت کرده و در زمان مناسب هر کدام اجرا می شوند.

## مقدمه ای درباره سیستم عامل و xv6

3.

- مدیریت حافظه اصلی
- مدیریت پردازنده، پردازش ها و فرآیندها
- مدیریت دستگاه های متصل به کامپیوتر و فایل ها

4.

### # basic headers

|                |                   |
|----------------|-------------------|
| 01 types.h     | Contains typedefs |
| 01 param.h     | Contains defines  |
| 02 memlayout.h | Contains defines  |

|           |  |
|-----------|--|
| 02 defs.h |  |
| 04 x86.h  |  |
| 06 asm.h  | Contains defines                                       |
| 07 mmu.h  | Contains defines and structs for x86 memory management |
| 09 elf.h  | Contains defines and structs for executable file       |
| 09 date.h | Contains struct for time                               |

## # entering xv6

|                 |   |
|-----------------|---|
| 10 entry.S      | The kernel starts executing in this file, this file is linked with the kernel C and transfers assembly symbols to |
| 11 entryother.S | Combines elements of bootasm.S and entry.S  |
| 12 main.c       | Starts operating system   |

## # locks

|               |
|---------------|
| 15 spinlock.h |
| 15 spinlock.c |

در صورتی که به صورت مشترک دو پردازنده به هر بخشی دسترسی بخواهند، با استفاده از lock میتوان آنرا مدیریت کرد که این مدیریت دسترسی سبب کاهش اختلال ها در کار هسته می شود.

## # processes

|             |
|-------------|
| 17 vm.c     |
| 23 proc.h   |
| 24 proc.c   |
| 30 swtch.S  |
| 31 kalloc.c |

این بخش پردازنده ها را مدیریت می کند.

(توابع مربوط به ساخت، ذخیره برای بازسازی یا reuseability و اختصاص حافظه فیزیکی به آن ها

## **# system calls**

32 traps.h

32 vectors.pl

33 trapasm.S

33 trap.c

35 syscall.h

35 syscall.c

37 sysproc.c

این بخش شامل trap ها و syscall می باشد. که توابع پیاده سازی شده مرتبط با انواع این دو ویژگی در فایل آنها موجود است.

## **# file system**

38 buf.h

39 sleeplock.h

39 fcntl.h

40 stat.h

40 fs.h

41 file.h

42 ide.c

44 bio.c

46 sleeplock.c

47 log.c

49 fs.c

58 file.c

60 sysfile.c

66 exec.c

استراکت ها و تعریف ها و توابعی برای مدیریت و ذخیره داده ها وجود دارد.

در bio.c کپی اطلاعات از دیسک به صورت linked list نگهداری می شود.

فایل log.c، یک logging ساده برای فراخوانی‌های همزمان سیستم می‌باشد.

در fs.c روتین‌های تغییرات سطح پایین برای فایل سیستم می‌باشد و پیاده‌سازی سطح بالای فراخوانی سیستم در فایل sysfile.c می‌باشد.

## # pipes

67 pipe.c

Pipe برای ایجاد ارتباط بین دو پردازش می‌باشد. وقتی پردازنده در حال مدیریت چند پردازش باشد توابع مربوطه با این قسمت این ارتباط را برقرار می‌کنند.

## # string operations

69 string.c

توابع مربوط به عملیات رشته‌ها

## # low-level hardware

70 mp.h

72 mp.c

73 lapic.c

76 ioapic.c

77 kbd.h

78 kbd.c

79 console.c

83 uart.c

این بخش مربوط به input output می‌باشد. Interface در این بخش قرار گرفته است و رابط کاربر با برنامه می‌باشد و به صورت کلی هر گونه تغییر یا درخواستی از سوی کاربر یا پردازش‌ها ایجاد شود این بخش آن‌ها را مدیریت می‌کند. همچنین ثابت‌های کیبورد نیز در این بخش هستند.

## # user-level

84 initcode.S

84 usys.S

85 init.c

85 sh.c

در اینجا برنامه‌های فضای کاربر را داریم.

در فایل `initcode.s` کدهایی برای اجرای `init.c` داریم، `init.c` برنامه اولیه در سطح کاربر است که اجرا می‌شود.

## # bootloader

91 bootasm.S

92 bootmain.c

بخش بوت مربوط به کدهای پایه‌ای برای اجرای سیستم عامل هستند که به صورت اسمبلی یا C نوشته شده است.

## # link

93 kernel.ld

اسکرپت ساده‌ای برای اینک کردن به هسته JOS است.

پوشه فایل‌های اصلی هسته سیستم عامل `boot` می‌باشد. پوشه فایل‌های سرایند کرنل `usr/src` و دیگر سرایندها `usr/include` می‌باشد.

نام پوشه فایل سیستم در سیستم عامل لینوکس / می‌باشد. (همان `root` است)

## کامپایل سیستم عامل xv6

5. بعد از اجرای دستور، متوجه up to date بودن xv6.img می‌شویم که نشان می‌دهد فایل نهایی هسته توسط xv6.img ساخته می‌شود.

6.

- UPROGS: یک متغیر است که یکسری دستور در آن وجود دارد که برای خوانایی و تمیزی کد وجود دارد.
- ULIB: یک متغیر است که فایل‌های ulib.o usys.o printf.o umalloc.o در آن قرار دارند که برای خوانایی و تمیزی کد وجود دارد.

7. پس از اجرای دستور خروجی زیر را گرفتیم:

```
$ alip@ap-laptop:~/UT/OS Lab/xv6-public-master$ make qemu -n
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
alip@ap-laptop:~/UT/OS Lab/xv6-public-master$
```

## اجرای بوت‌لودر

8. با توجه به خروجی دستور make -n، فایل xv6.img را می‌بینیم، حال در فایل makefile در می‌یابیم که xv6.img به bootblock و kernel وابسته می‌باشد. پس از مراجعه به bootblock در makefile، دو فایل bootasm.S و bootmain.c را می‌بینیم، بنابراین در سکتور نخست قابل بوت محتوای این دو فایل قرار دارد.

9.

1. نوع این فایل دودویی o. یا همان object است.
2. تفاوت این فایل با دیگر فایل‌ها این است که سیستم عامل آن را تولید کرده تا برای زبان ماشین قابل فهم باشد. به عبارتی هر خط از کد ترجمه شده و به دستورات CPU تبدیل می‌شود. اما سایر فایل‌های دودویی یا توسط کاربر و یا توسط قسمتی از برنامه تولید شده و مستقلاً این فایل‌ها قابل استفاده هستند و ترجمه شده از فایل دیگری نیستند.
3. این نوع فایل دودویی بعد کامپایل تولید می‌شود. که البته کنار object files فایل دیگری با عنوان linker files نیز وجود دارد.

Linker files فایل‌های دودویی تولید شده را به یک فایل قابل اجرا توسط ماشین تبدیل می‌کنند. درواقع کدی که ما مینویسیم زبانی است که انسان‌ها می‌فهمند و برای ما قابل فهم است. ماشین‌ها با زبانی دودویی کار میکنند پس باید فایل‌ها را به زبان خودشان ترجمه کرده، تا در لایه‌های سطح پایین ماشین قابل فهم باشد.

10. فایل های object را کپی و ترجمه می کند. Objcopy برای ترجمه فایل های object، فایل های موقتی می سازد و سپس آن ها را پاک می کند.

11. به طور کلی زبان های C و اسمبلی، زبان های نزدیک تر به زبان ماشین هستند و وقتی نیاز به دسترسی به سطوح لایه های پایینی سیستم عامل داریم، بهتر است از این زبان ها به جای زبان های پیشرفته تر مثل پایتون یا C# استفاده شود تا سرعت اجرای کد و بوت شدن سیستم تسریع شود. کد اسمبلی نیز چون سطح پایین تری دارد پس با تابع bootmain رابط بین پردازنده و کد C می باشد.

12.

**ثبات عام منظوره: AX** که برای عملیات های محاسباتی است

**ثبات قطعه: SS** پوینتر به استک

**ثبات وضعیت: ZERO flag** نشان می دهد که اگر نتیجه یک عملیات محاسباتی صفر است یا خیر

**ثبات کنترلی: WP** اگر فعال باشد CPU نمی تواند روی فایل های readonly چیزی بنویسد

13. چون از نسخه قبلی به ارث رسیده بود و باگ های قبلی برطرف شده بود تا از قابلیت protected mode هم استفاده شود. نقص آن این بود که فقط در صفحه startup فعال بود و هسته CPU را به حالت long mode می برد.

14.

**16-bit real mode: bit** بخش ها دارای آدرسی دهی مستقیم 16 بیتی هستند

**16-bit protected mode: bit** بخش ها دارای ایندکس هایی هستند که با مراجعه به جدول می توان به آن ها دسترسی پیدا کرد.

**32-bit protected mode: bit** مشابه بالایی فقط حافظه 32 بیتی است

**64-bit long mode and 32-bit** این حالت به دستورات 32 بیتی و رجیستر ها می تواند دسترسی داشته باشد

15.

$$\text{Offset} + \text{PhysicalAddress} = \text{Segment} * 16$$

16. انتظار می‌رفت هسته در آدرس 0x80100000 قرار دهد ولی به دلیل آنکه ممکن است حافظه فیزیکی این آدرس که بزرگ است در ماشین کوچک وجود نداشته باشد و دلیلی که هسته را در آدرس 0x0 قرار نمی‌دهد آن است که بازه آدرس 0xa0000:0x100000 می‌باشد.

17. هسته ابتدا در `startup_32` در `arch/x86/kernel/head_32.S` شروع به کار می‌کند. سپس کد با صدا زدن `i386_start_kernel()` در `C` در `arch/x86/kernel/head32.C` تمام می‌شود. سپس برای راه اندازی هسته لینوکس `start_kernel()` را فراخوانی می‌کند.

18. چون اگر تغییری ایجاد شود، این تغییر به صورت گلوبال بر روی BIOS نیز اعمال میشود و شاید می‌نخواهیم این تغییرات گلوبال صورت گیرد پس با استفاده از `entry.s` بدون نیاز به ایجاد این تغییرات عملیات را انجام داده و دسترسی هسته را از عملیات قطع می‌کنیم.

## اجرای هسته xv6

19. استفاده از آدرس مجازی بهینه نیست. چراکه برای ذخیره هر آدرس مجازی نیاز به یک ذخیره فیزیکی می‌بود که در این صورت به ازای هر آدرس، هم مجازی نداشتیم هم فیزیکی.

20. تابع ابتدایی برای بوت شدن سیستم است. سپس آدرس شروع به صورت فیزیکی استخراج می‌شود. حال بوت شدن دوم را داریم که با توجه به آدرس فیزیکی دریافت شده صفحه مورد نظر بوت می‌شود. حال اندازه و ابعاد این صفحه مقدار میگیرد و دایرکتوری آن نیز قرار داده می‌شود. در نهایت پویتر به استک تعریف میشود. در نهایت نیز به `main` منتقل میشویم. `start_kernel()` تابع معادل است.

21. سیستم عامل شامل 32 بین فضای مجازی آدرس است. 2 گیگابایت زیرین مربوط به عملیات از سوی کاربر است (1 گیگ آن برای سیستم و 1 گیگ برای کد کاربر، استک، داده و هیپ). همچنین 2 گیگابایت بالایی مخصوص هسته می‌باشد. بنابراین هم کاربر و هم هسته می‌توانند از این فضا استفاده کرد.

22. گاهی در فرایند ترجمه دو دستوراعمل تقریباً مشابه (مثلاً فقط در یک مولفه خاص تفاوت دارند) ترجمه یکسان دارند و لزوماً یکتا نمیشوند. به همین خاطر برای تفکیک این حالت از پرچم استفاده می‌کنیم.



23. ساینز حافظه مورد نیاز برای پردازش را داریم.

استک و همچنین جدول آدرس صفحات را به عنوان پوینتر ذخیره می کنیم.

سپس حالت پردازش و ای دی یکتا این پردازش ذخیره می شود.

پدر این پردازش به صورت پوینتر داده می شود (بالا تر راجب بچه پردازش ها توضیح داده بودم)

دو پرچم خواب و کشتن داریم.

سپس دو استراکت یکی برای باز کردن فایل و دیگری برای پیدا کردن دایرکتوری تعریف شده است.

در نهایت اسم این پردازش به صورت رشته ذخیره شده است که برای دیباگ کردن قابل استفاده است.

معادل این استراکت که عملیات مربوط به `pre process` را به عهده دارد، در هسته لینوکس به صورت زیر است:

```
KERNELDIR = /usr/src/linux
```

```
config/(KERNELDIR))$ include
```

```
O -Wall- \include/(KERNELDIR))$CFLAGS = -D__KERNEL__ -DMODULE -I
```

```
CONFIG_SMP ifdef
```

```
CFLAGS += -D__SMP__ -DSMP
```

```
endif
```

```
all: skull.o
```

```
skull.o: skull_init.o skull_clean.o
```

```
@$ o- ^$ r-LD))$
```

```
clean:
```

```
rm -f *.o *~ core
```

24. از آنجایی برنامه های سطح کاربر و اجرای پوسته زمان بندی می شوند، در صورت به خواب رفتن کد مدیریت کننده سیستم، کد های

آماده سازی اجرا شده و این باعث بروز اختلال در برنامه می شود. چون باید مجدداً بررسی شود ادامه اجرای سطح هسته از کجا رخ می

دهد.

## اجرای نخستین برنامه سطح کاربر

25. `Kvmmalloc()` برای فضای آدرس هسته است که به صورت خودراه انداز پردازش می شود. این فضای هسته با استفاده از یک استک

کار می کند که قابلیت سوییچ کردن از این فضا به آن فضای پشته همواره وجود دارد. همچنین این فضا به حافظه نیز تخصیص داده شده

است. اما فضای `setupkvm()` متفاوت است. این فضای آدرس جامع تری را می تواند دسترسی داشته باشد. در واقع `Kvmmalloc()`

نیز خودش برای دسترسی به حافظه مربوطه ابتدا `setupkvm` را صدا می زند و پوینتری از خروجی را نگه می دارد.

فضای دسترسی `setupkvm` به صورت زیر است:

x86 using 4KB pages

virtual address 0x80100000 to physical address 0x00100000 with read-only permissions

و برای دسترسی به این صفحات به افسر که بالاتر توضیح داده شده نیاز است.

.26

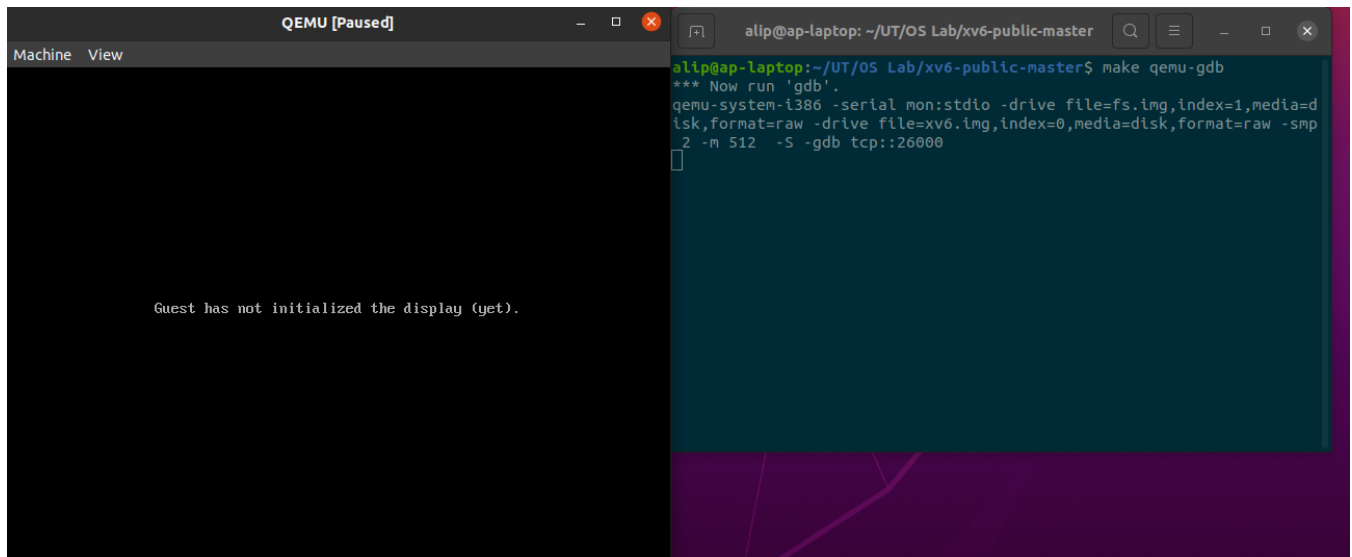
27. هنگامی که `exec()` صدا می‌شود، فایل اجرایی قبلی جایگزین شده و فایل جدیدی اجرا می‌شود. در کل `exec` فایل یا برنامه قدیمی در حال پردازش را با فایل یا برنامه جدید جایگزین می‌کند.

.28

```
char init[] = "/init\0";
char *argv = {init, 0};
exec(init, argv);
for(;;) exit();
```

## اشکال زدایی

با وارد کردن دستور `make qemu-gdb` داریم:



پس از اجرای دستورات gdb\_cat و target remote tcp::26000 داریم:

```
alip@ap-laptop: ~/UT/OS Lab/xv6-public-master
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from _cat...
warning: File "/home/alip/UT/OS Lab/xv6-public-master/.gdbinit" auto-loading has
been declined by your `auto-load safe-path' set to "$debugdir:$datadir/auto-loa
d".
To enable execution of this file add
    add-auto-load-safe-path /home/alip/UT/OS Lab/xv6-public-master/.gdbinit
line to your configuration file "/home/alip/.gdbinit".
To completely disable this security protection add
    set auto-load safe-path /
line to your configuration file "/home/alip/.gdbinit".
For more information about this security protection see the
--Type <RET> for more, q to quit, c to continue without paging--c
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
    info "(gdb)Auto-loading safe path"
(gdb) target remote tcp::26000
Remote debugging using tcp::26000
0x0000ffff in ?? ()
(gdb)
```

## روند اجرای GDB

1. با دستور `info breakpoints` جدولی از تمام breakpointها چاپ می‌شود.
2. می‌توان پس از اجرای دستور `info breakpoints` و چاپ breakpointها، با دستور `del num` آن breakpoint مختص به آن num را پاک کرد. همچنین می‌توان با دستور `clear filename:linenum` اسم فایل و شماره خط را داده و breakpoint آن خط را پاک کند.

## کنترل روند اجرا و دسترسی به حالت سیستم

3. Backtrace خلاصه‌ای از این است که برنامه چگونه به این مرحله رسیده است. همچنین یک خط به ازای فریم در حال اجرا نمایش می‌دهد و در خط دیگر تابع فراخوانی شده را نشان می‌دهد. در خط‌های بعدی همه‌ی فریم‌های موجود در استک را پرینت می‌کند.

```
alip@ap-laptop: ~/UT/OS Lab/xv6-public-master
line to your configuration file "/home/alip/.gdbinit".
For more information about this security protection see the
--Type <RET> for more, q to quit, c to continue without paging--
"Auto-loading safe path" section in the GDB manual. E.g., run from the shell:
    info "(gdb)Auto-loading safe path"
(gdb) target remote tcp::26000
Remote debugging using tcp::26000
0x0000ffff in ?? ()
(gdb) break cat
Breakpoint 1 at 0x90: cat. (2 locations)
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, cat (fd=0) at cat.c:9
9      {
(gdb) n
printf (fd=1, fmt=<optimized out>) at printf.c:51
51      if(c == '%'){
(gdb) n
63      } else if(c == 's'){
(gdb) bt
#0  printf (fd=1, fmt=<optimized out>) at printf.c:63
#1  0x00000097 in cat (fd=0) at cat.c:9
#2  0x0000089f in ?? ()
```

4. دستور x محتویات یک آدرس حافظه را نشان می دهد اما دستور print مقدار ذخیره شده در یک متغیر یا عبارت را نشان می دهد.

```
(gdb) print stoi
No symbol "stoi" in current context.
(gdb) print cat
$1 = {void (int)} 0x90 <cat>
(gdb) x 0x00000123
0x123 <strcpy+19>: 0xc4830000
(gdb)
```

5. برای نمایش وضعیت ثبات ها می توان از دستور info registers استفاده کرد.

```
alip@ap-laptop: ~/UT/OS Lab/xv6-public-master
(gdb) x 0x00000123
0x123 <strcpy+19>: 0xc4830000
(gdb) info registers
eax             0x2fc8             12232
ecx             0x33              51
edx             0x0               0
ebx             0x33              51
esp             0x2f80            0x2f80
ebp             0x2fb8            0x2fb8
esi             0x8a0             2208
edi             0x2f9f            12191
eip             0x551             0x551 <printf+129>
eflags          0x246             [ IOPL=0 IF ZF PF ]
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             0xdfbb000         [ PDBR=0 PCID=0 ]
cr4             0x10             [ PSE ]
--Type <RET> for more, q to quit, c to continue without paging--
```

برای متغیرهای محلی باید از دستور `info locals` استفاده کرد.

```
(gdb) info locals
s = <optimized out>
c = 12232
i = <optimized out>
state = 0
ap = 0x2fc8
(gdb)
```

برای نمایش یک ثابت خاص نیز می‌توان از دستور `info register registername` استفاده کرد.

```
(gdb) info register cs
cs                0x1b                27
```

Si = source index

آدرس خواندن همه عملیات مربوط به رشته‌ها را نگه می‌دارد

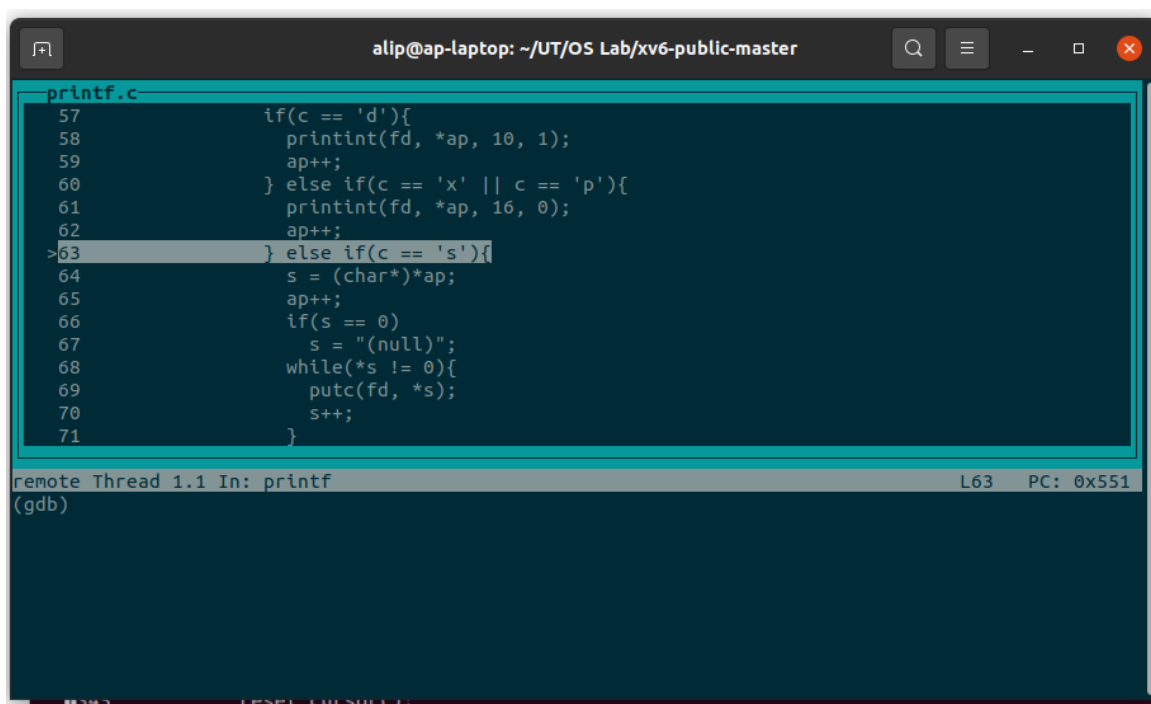
Di = destination index

آدرس نوشتن همه عملیات مربوط به رشته‌ها را نگه می‌دارد

.6

## اشکال‌زدایی در سطح اسمبلی

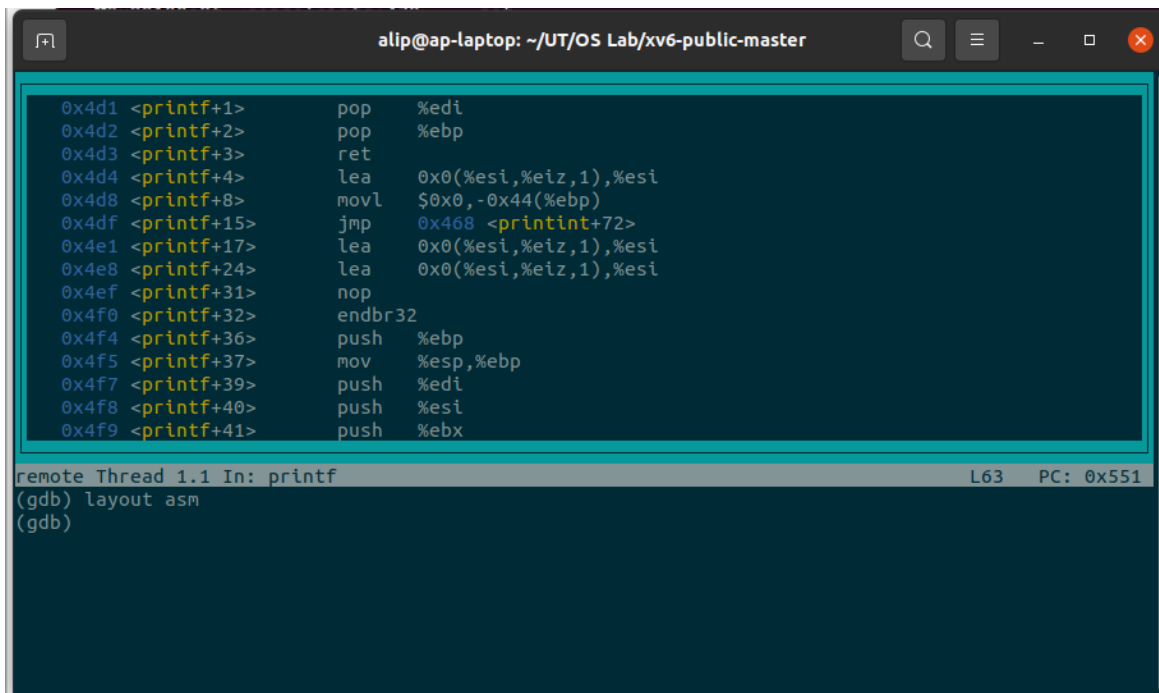
7. عکس زیر مربوط به اجرای دستور `src layout` می‌باشد.



```
printf.c
57         if(c == 'd'){
58             printint(fd, *ap, 10, 1);
59             ap++;
60         } else if(c == 'x' || c == 'p'){
61             printint(fd, *ap, 16, 0);
62             ap++;
63         } else if(c == 's'){
64             s = (char*)*ap;
65             ap++;
66             if(s == 0)
67                 s = "(null)";
68             while(*s != 0){
69                 putc(fd, *s);
70                 s++;
71             }
>0x551:    jne     0x551<+1>

remote Thread 1.1 In: printf
(gdb)
```

عکس زیر مربوط به اجرای دستور `layout asm` می‌باشد.

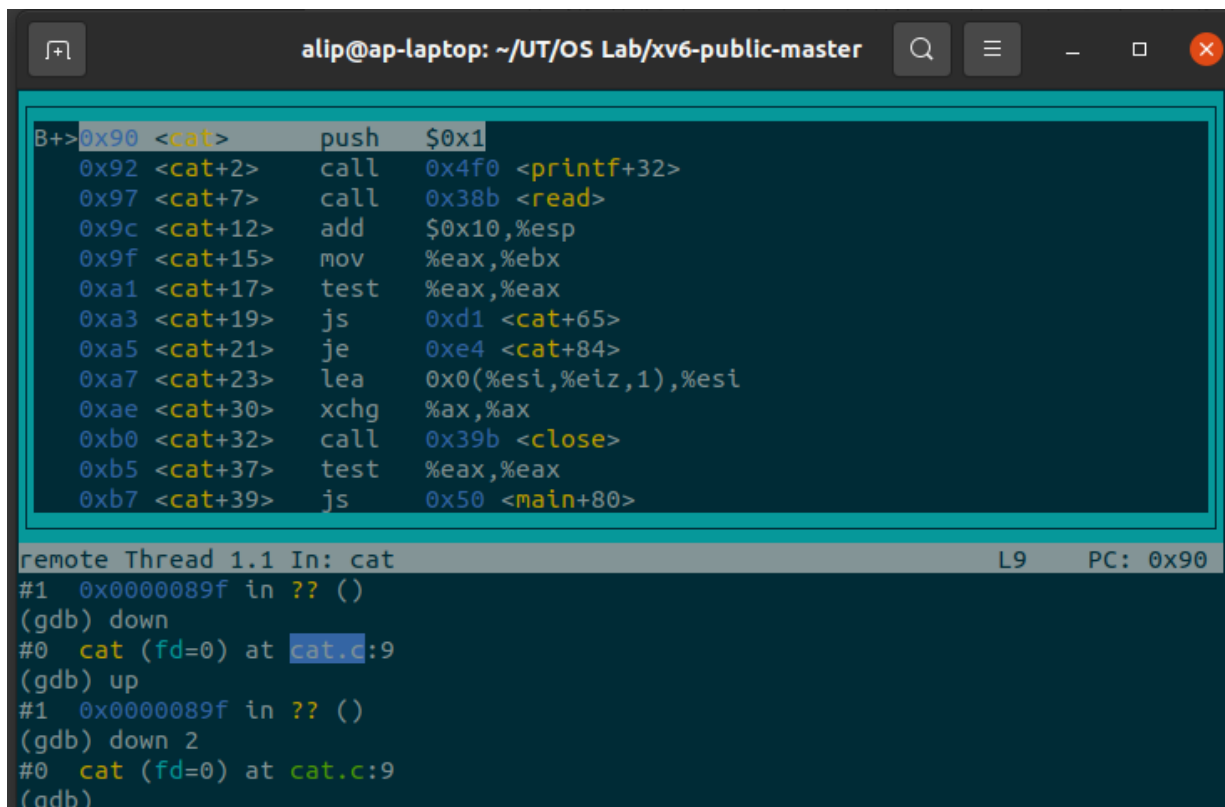


```
alip@ap-laptop: ~/UT/OS Lab/xv6-public-master

0x4d1 <printf+1>    pop    %edi
0x4d2 <printf+2>    pop    %ebp
0x4d3 <printf+3>    ret
0x4d4 <printf+4>    lea    0x0(%esi,%eiz,1),%esi
0x4d8 <printf+8>    movl   $0x0,-0x44(%ebp)
0x4df <printf+15>   jmp     0x468 <printint+72>
0x4e1 <printf+17>   lea    0x0(%esi,%eiz,1),%esi
0x4e8 <printf+24>   lea    0x0(%esi,%eiz,1),%esi
0x4ef <printf+31>   nop
0x4f0 <printf+32>   endbr32
0x4f4 <printf+36>   push   %ebp
0x4f5 <printf+37>   mov    %esp,%ebp
0x4f7 <printf+39>   push   %edi
0x4f8 <printf+40>   push   %esi
0x4f9 <printf+41>   push   %ebx

remote Thread 1.1 In: printf                                L63  PC: 0x551
(gdb) layout asm
(gdb)
```

8. برای جابجایی میان توابع زنجیره فراخوانی جاری از دستورهای `up` و `down` استفاده می‌شود که جلوی آنها می‌توان تعداد پرش‌ها را نیز مشخص کرد که حالت پیش‌فرض آن یک واحد می‌باشد.



```
alip@ap-laptop: ~/UT/OS Lab/xv6-public-master

B+> 0x90 <cat>    push   $0x1
0x92 <cat+2>      call   0x4f0 <printf+32>
0x97 <cat+7>      call   0x38b <read>
0x9c <cat+12>     add     $0x10,%esp
0x9f <cat+15>     mov     %eax,%ebx
0xa1 <cat+17>     test    %eax,%eax
0xa3 <cat+19>     js      0xd1 <cat+65>
0xa5 <cat+21>     je      0xe4 <cat+84>
0xa7 <cat+23>     lea     0x0(%esi,%eiz,1),%esi
0xae <cat+30>     xchg    %ax,%ax
0xb0 <cat+32>     call   0x39b <close>
0xb5 <cat+37>     test    %eax,%eax
0xb7 <cat+39>     js      0x50 <main+80>

remote Thread 1.1 In: cat                                    L9   PC: 0x90
#1  0x0000089f in ?? ()
(gdb) down
#0  cat (fd=0) at cat.c:9
(gdb) up
#1  0x0000089f in ?? ()
(gdb) down 2
#0  cat (fd=0) at cat.c:9
(gdb)
```