



OPERATING SYSTEM LAB

SESSION 1

| | |
|-----------------------------|-----------|
| SayedMehdi HajiSayedHossein | 810100118 |
| Alireza Hosseini | 810100125 |
| AmirAli Shahriary | 810100173 |

سوال ۱ :

معماری سیستم عامل **xv6** چیست؟ چه دلایلی در دفاع از نظر خود دارید؟
این سیستم عامل بر اساس معماری که **unix version 6 -ANSI C** نوشته شده است و معماری و ساختار های مشابهی با یکدیگر دارند که مبتنی بر پردازنده های **X86 multiprocessor** به منظور استفاده در اهداف آموزشی پیاده سازی شده است.

با توجه به اینکه می دانیم معماری **x86** از **unix** پیروی می کند از دلایل آن می توان به :

- ذکر استفاده از معماری **x86** در فایل **asm.h** .
- در فایل **trap.h** از **trap** های مختص معماری **86x** استفاده شده است.
- فایل **xh6.h** که منحصر از دستورات اسمبلی مختص این معماری و پردازنده استفاده شده است.
- فایل **asm.h** نیز مجدد از همین معماری بهره برده است.
- همچنین در فایل منابع سیستم عامل (**11book-rev**) صراحتا به استفاده از معماری (**intel 80386**) (**86x or later**) اشاره کرده است.

سوال ۲ :

هر برنامه در حال اجرا یک **process** (پردازه) نام دارد . که از بخش های

۱. حافظه ی شامل دستورات (memory containing instructions)

۲. دیتا های مربوط پردازه (variables on which the computation acts)

۳. یک استک (The stack organizes the program's procedure calls) تشکیل شده است .

زمانی که **process** یک سرویس کرنل را درخواست می کند از طریق **system calls** آن را به کرنل می رساند. کرنل حالا این سرویس لازم را اجرا میکند و جواب را بازمیگرداند از این طریق ارتباط بین فضای کاربر و فضای کرنل برقرار میشود . [user space / kernel space]

حالا کرنل با استفاده از **CPU's hardware protection mechanisms** مطمئن میشود که هر **process** در **user space** فقط به مموری مربوط به خودش دسترسی دارد . کرنل این کار هارا با **privileges access** انجام میدهد که این کار توسط **user program** ها میسر نیست .

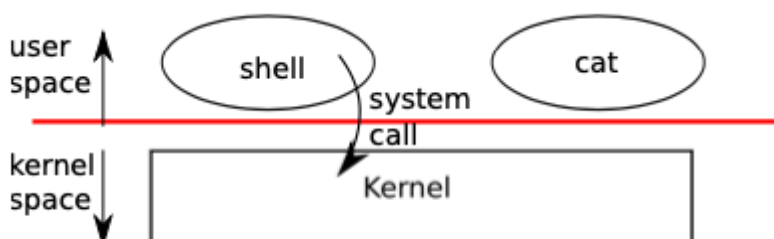


Figure 0-1. A kernel and two user processes.

سوال ۴ :

- Fork System Call:

یک **process** میتواند با استفاده از سیستم کال **fork** یک پروسس فرزند بسازد (**child process**) که دقیقا همان محتوای مموری **process** پدر را دارد .
مقدار خروجی **fork** برای پروسس پدر (**process identifier (pid)**) فرزند است و برای پروسس فرزند مقدار 0 میباشد .

- Exec System Call:

این سیستم کال مموری پروسس پدر را با یک **memory image** جدید که از سیستم فایل قابل دسترسی است جایگزین میکند . این فایل ها شامل **instruction** های مربوط به آن دستور اند که با فرمت **ELF** هستند . اگر سیستم کال **exec** موفق به اجرا شود چیزی بر نمیگرداند به پروسس پدر در عوض دستورات از فایل **elf** لود میشوند و شروع به اجرا میکنند .
این سیستم کال دو ورودی دارد :

۱ . آدرس فایل **ELF**

۲ . آرایه ای **str** که شامل آرگومان های ورودی دستور خاص لود شده توسط فایل **ELF** است .

ادامه سوال ۴ : ادغام نکردن این دو چه مزیتی داره ؟

سیستم کال **fork** مقدار حافظه ای که برای پروسس فرزند از روی حافظه ی پدر لازم است را اختصاص میدهد اما **exec** به مقدار کافی فقط برای نگه داشتن **executable file** حافظه تخصیص میدهد .
یک نکته دیگر هم که هست اینکه :

```
char *argv[2];  
  
argv[0] = "cat";  
argv[1] = 0;  
if(fork() == 0) {  
    close(0);  
    open("input.txt", O_RDONLY);  
    exec("cat", argv);  
}
```

Because if they are separate, the shell can fork a child, use open, close, dup in the child to change the standard input and output file descriptors, and then exec. No changes to the program being exec-ed (cat in our example) are required. If fork and exec were combined into a single system call, some other (probably more complex) scheme would be required for the shell to redirect standard input and output, or the program itself would have to understand how to redirect I/O.

سوال ۸ :

در Makefile متغیر هایی به نام UPROGS و ULIB تعریف شده است. کاربرد آنها چیست؟

متغیر : UPROGS

UPROGS که به اختصار User Programs نوشته می شود . این متغیر حامل لیستی از برنامه های کاربر است که در سیستم عامل موجود است و در هنگام کامپایل تبدیل به فایل های قابل اجرا توسط سیستم عامل می شوند. برخی از برنامه ها از پیش موجود هستند و برای برنامه های جدید نام هریک را تک تک به آن اضافه می کنیم (دقیقاً مشابه کاری که برای افزودن برنامه سطح کاربر `strdiff` انجام دادیم) . فایل آبجکت برنامه های کاربر که در `uprogs` ایجاد شدند ، در نهایت منجر به اجرای دستور `ld` شده که این دستور برای پیوند فایل های مورد نیاز و تولید یک فایل قابل اجرا مورد استفاده قرار می گیرند. از طرفی باید بدانیم که آبجکت های هر برنامه (O.) بنا به یک قانون درونی در `MAKEFILE` ساخته می شوند اما به صورت صریح در آنجا نوشته نمی شوند.

متغیر :ULIB

ULIB که به اختصار User Libraries نوشته می شود ، همانطور که از نامش پیداس حامل کتابخانه هایی به زبان C بوده که در کد های سیستم عامل از آن استفاده شده است و برای اجرای آنها طبیعتاً نیازمندیم که آنها کامپایل شوند. این فایل ها به عنوان پیشنیاز در قوانین قرار گرفته اند و در آخر توسط `ld` به فایل اجرایی پیوند می شوند. این فایل شامل توابع ابتدایی چون `printf - malloc - strcpy` هستند.

سوال ۱۱ :

بوت لودر `xv6` از دو سورس فایل است . اولی تریکت `16-bit` و `32-bit` برای `x86` اسمبلی که در فایل `bootasm.S` است و دیگری که در فایل `bootmain.c` نوشته شده است .

The C part of the boot loader, `bootmain.c` (9200), expects to find a copy of the kernel executable on the disk starting at the second sector. The kernel is an ELF format binary, as we have seen in Chapter 2. To get access to the ELF headers, `bootmain` loads the first 4096 bytes of the ELF binary (9214). It places the in-memory copy at address `0x10000`

با دستور `objdump -d filename`

```
root@UBUNTU: /tmp/xv6-public
objdump
Usage: objdump <option(s)> <file(s)>
Display information from object <file(s)>.
At least one of the following switches must be given:
-a, --archive-headers    Display archive header information
-f, --file-headers       Display the contents of the overall file header
-p, --private-headers    Display object format specific file header contents
-P, --private=OPT,OPT... Display object format specific contents
-h, --[section-]headers  Display the contents of the section headers
-x, --all-headers        Display the contents of all headers
-d, --disassemble        Display assembler contents of executable sections
-D, --disassemble-all   Display assembler contents of all sections
    --disassemble=<sym>  Display assembler contents from <sym>
-S, --source             Intermix source code with disassembly
    --source-comment[=<txt>] Prefix lines of source code with <txt>
-s, --full-contents      Display the full contents of all sections requested
-g, --debugging          Display debug information in object file
-e, --debugging-tags     Display debug information using ctags style
-G, --stabs              Display (in raw form) any STABS info in the file
-W, --dwarf[a/=abbrev, A/=addr, r/=aranges, c/=cu_index, L/=decodedline,
    f/=frames, F/=frames-interp, g/=gdb_index, i/=info, o/=loc,
    m/=macro, p/=pubnames, t/=pubtypes, R/=Ranges, l/=rawline,
    s/=str, O/=str-offsets, u/=trace_abbrev, T/=trace_aranges,
    U/=trace_info]       Display the contents of DWARF debug sections
-wk, --dwarf=links       Display the contents of sections that link to
    separate debuginfo files
-WK, --dwarf=follow-links Follow links to separate debug info files (default)
-WN, --dwarf=no-follow-links Do not follow links to separate debug info files
-L, --process-links      Display the contents of non-debug sections in
    separate debuginfo files. (Implies -WK)
    --ctf[=SECTION]      Display CTF info from SECTION, (default '.ctf')
-t, --syms               Display the contents of the symbol table(s)
-T, --dynamic-syms       Display the contents of the dynamic symbol table
-r, --reloc              Display the relocation entries in the file
    --dynamic-reloc      Display the dynamic relocation entries in the file
```

سوال ۱۲ :

علت استفاده از دستور objcopy در حین اجرای عملیات make چیست ؟

همانگونه که از نام دستور مشخص است این دستور محتویات یک آبجکت را کپی کرده و در فایل آبجکت دیگری ذخیره می کند. که لزومی هم به یکسان بودن فرمت مقصد و آن وجود ندارد. این ترجمه به کمک کتابخانه GNU BFD انجام شده و به صورت داخلی تمامی فرمت های ممکن پشتیبانی می شود و تبدیل پذیری تایپ ها به سادگی انجام می پذیرد. عملکرد objcopy توسط کاربر و دستور ورودی او در ترمینال مشخص می شود این دستور برای ترجمه فایل های object از فایل های موقت (temp) استفاده می کند و در ادامه آنها را پاک می کند. در بخش هایی از makefile از آن استفاده شده که به تشریح آن می پردازیم.

- در initcode محتویات فایل out.initcode در یک فایل binary raw کپی شده که در ادامه با لینک شدن فایل های o.entry های و فایل object که در متغیر OBJS پیشتر تعریف شده اند و فایل های باینری initcode و entryother که پیشتر با استفاده از دستور objcopy ساخته شدند، فایل kernel ایجاد می شود.
- در bootblock پس از لینک شدن bootmain.o و bootasm.o در bootblock.o محتویات txt فایل فوق را در فایل binary raw به نام bootblock کپی کرده و در نهایت این فایل را به اسکریپت sign.pl که با بررسی ساینز فایل boot signature را به انتهای آن اضافه می کند.
- در entryother محتویات بخش text فایل bootblockother.o را در یک فایل binary raw به نام entryother کپی می کند.

همچنین آپشن هایی از این دستور بنا به معماری xv6 در makefile که استفاده می شوند را نیز مورد بررسی قرار می دهیم:

- آپشن -s: حرف s که عملاً اول حرف symbol table است اطلاعات آنرا اعم از نام و آدرس متغیر ها را سیو کرده که این احتمالاً در دیگر فایل های obj نیز مورد استفاده قرار گرفته است. و اطلاعات مربوط به symbol table و relocation table در فایل مقصد حذف می شوند. Relocation table حاوی اطلاعاتی از مکان فایل های obj مورد استفاده است که در زمان ساخت فایل معلوم نبوده ولی در ادامه بایستی توسط لینکر مقدار دهی شوند.
- آپشن -J: بخشی از فایل obj به فایل جدید کپی می شود.
- آپشن -O: بیانگر فرمت فایل مقصد می باشد مثلاً برای O BINARY- بیانگر raw binary است. این تیپ از فایل ها به فرمت خاصی نوشته نشده.

سوال ۱۴:

یک ثبات عام منظوره، یک ثبات قطعه، یک ثبات وضعیت و یک ثبات کنترلی در معماری xv6 را نام برده و وظیفه هر یک را به طور مختصر توضیح دهید.

```
(gdb) info registers
eax            0x0                0
ecx            0x0                0
edx            0x663             1635
ebx            0x0                0
esp            0x0                0x0
ebp            0x0                0x0
esi            0x0                0
edi            0x0                0
eip            0xffff0            0xffff0
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--
cr8            0x0                0
```

General purpose REG :

این رجیسترها برای کارهای مختلفی در معماری قابل استفاده اند. مثل: **eax** که برای محاسبات ریاضی و مقدار بازگشتی برای فانکشن کال ها استفاده میشود **ebx** برای پوینتر به دیتا استفاده میشود و **ecx** که برای شمارش لوپ معمولا استفاده میشود

Segment REG :

این رجیسترها برای منیج و مدیریت کردن و دسترسی به بخش های مختلف حافظه اند. مثل **CS** این رجیستر حاوی پوینتری است که باید دستورات از آنجا شروع شود. **SS** این رجیستر برای استک سگمنت است. برای فانکشن کال ها و پوش و پاپ از استک مورد استفاده است.

Status REG:

در معماری **x86** مستقیما چیزی به اسم **status register** نداریم، بلکه **flag** هایی هستند که **EFLAGS** نام دارند که وضعیت اپراتور ها را مشخص میکنند.

{ **zero FLAG** ← ریزالت صفر به این رجیستر مراجعه میکند }

{ **sign flag** ← علامت مثبت یا منفی ریزالت رو مشخص میکند }

Control REG :

این رجیستر ها برای کنترل کردن رفتار پردازنده ها (**process** ها) استفاده میشوند. **[0CR]** برای کنترل کردن برای پرتکشن نکانیستم **user mode** , **kernel mode** است **[2CR]** برای ذخیره کردن ادرس آخرین **page fault**

سوال ۱۸ : کد معادل **entry.s** را در هسته لینوکس را بیابید .

ارجاع به حساب لینوس توروالدز - کد متناظر در لینک فوق :

<https://github.com/torvalds/linux/blob/master/arch/arm64/kernel/entry.S>

سوال ۱۹: چرا این آدرس فیزیکی است؟

اگر این بخش مجازی بود آنگاه مجدداً نیازمند یک بخش فیزیکی بودیم تا بخش مجازی فوق را مشخص کند پس انگار در نهایت باز به بخش فیزیکی نیاز داشتیم و عملاً این ممکن نبود و تناقض رخ می دهد. یا به عبارتی دیگر با نیازمندی مجدد به خودش برای یافتن آدرس فیزیکی آن خله ، لوپ بی نهایت ایجاد شده و مدام اجرا میشود و برای برگ زدن به این لوپ نیازمند آدرس فیزیکی خواهیم بود.

سوال ۲۲: چرا برای کد و داده های سطح کاربر پرچم **SEG_USER** تنظیم شده است؟

همانطور که در فایل mmu.h تعریف SEG مشاهده می شود :

```
// Normal segment
#define SEG(type, base, lim, dpl) (struct segdesc) \
{ ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff, \
  ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1, \
  (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
```

و همچنین در فایل VM.C قطعه بندی سیستم عامل در seginit قطعه کد زیر مشاهده می شود:

```
void
seginit(void)
{
    struct cpu *c;

    // Map "logical" addresses to virtual addresses using identity map.
    // Cannot share a CODE descriptor for both kernel and user
    // because it would have to have DPL_USR, but the CPU forbids
    // an interrupt from CPL=0 to DPL=3.
    c = &cpus[cuid()];
    c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
    c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
    c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
    c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
    lgdt(c->gdt, sizeof(c->gdt));
}
```

بنا به توضیحات شرح آزمایش هر قطعه که بخشی از حافظه را اشغال نموده است با دسکریپتوری مشخص با Global descriptor table ارتباط دارند که شامل اطلاعاتی از قطعه مانند اندازه و سطح دسترسی و غیره هستند.

در توضیح مراحل خواندن یک اینستراکشن نخست توسط دسکریپتور آن یافت می گردد و آدرس منطقی آن در صفحه یافت شده به آدرس فیزیکی تبدیل می گردد و اینستراکشن آن خوانده و اجرا می شود. وقتی مکان قطعه در دسکریپتور قطعه مشخص می شود سطح دسترسی CPL (current privilege level) و برابر سطح دسترسی دسکریپتور DPL Descriptor privilege level است که DPL های متفاوت قادرند سطح دسترسی فعلی دستور العمل هارا نیز تعیین کرد حتی اگر قطعات یکسانی را تعریف کنند.

سوال ۲۳: جهت نگهداری اطلاعات مدیریتی برنامه های سطح کاربر ساختاری تحت عنوان **struct** ارایه شده است. **proc** اجزای آن را توضیح دهید و ساختار معادل آن در سیستم عامل لینوکس را بیابید

- **SZ:** سایز حافظه متعلق به پردازنده byte
- **pgdir:** پوینتر به page table
- **kstack:** پایین استک کرنل برای این پردازنده را مشخص می کند عملاً پوینتر به کرنل استک به منظور اجرای system call ها استفاده می گردد.
- **state:** وضعیت پردازنده فوق را مشخص می کند ، وضعیت هایی چون zombie , running & etc
- **pid:** عدد مختص پردازنده (یکتا) .
- **parent:** والد پردازنده فوق یا همان سازنده آن است.
- **tf:** چارچوب مشخص برای trap برای system call حاضر.
- **context:** نگهداری به منظور context switching
- **chan:** به اختصار channel است و اگر 0 نباشد به معنی خوابیدن پردازنده است.
- **killed:** اگر غیر صفر باشد به منزله kill شدن پردازنده است.
- **ofile:** فایل های باز شده توسط این پردازنده.
- **cwd:** بیانگر فولدر یا دایرکتوری کنونی .
- **name:** نمایانگر نام پردازنده .

ساختار معادل آن در سیستم عامل لینوکس sched.h در کرنل لینوکس است که عملاً معادل استراکت است.

برای دسترسی آن از حساب گیت هاب خالق هسته کرنل لینوکس ، لینوس توروالدز به آن دسترسی پیدا می کنیم:

<https://github.com/torvalds/linux/blob/master/include/linux/sched.h>

سوال ۲۷:

هسته اول که فرآیند بوت را انجام میدهد توسط کد **entry.S** وارد تابع **main** در فایل **main.c** می شود. تمامی توابع آماده سازی سیستم که در این تابع فراخوانده شدهاند توسط این هسته اجرا شوند.

بخش‌های مشترک در تمامی هسته‌ها هنگام آماده‌سازی سیستم :

- switchkvm
- seginit
- lapicinit
- mpmain

بخش‌های اختصاصی هسته اول :

- kinit
- kvmalloc
- setupkvm
- mpinit
- consoleinit
- uartinit
- picinit
- ioapicinit
- pinit
- tvinit
- binit
- fileinit
- ideinit
- startothers
- kinit2
- userinit

بخش اشکال زدایی

۱ - برای مشاهده Breakpoint ها از چه دستوری استفاده می شود؟

با استفاده از دستور `info b`

```
(gdb) b console.c:50
Breakpoint 3 at 0x80100678: file console.c, line 51.
(gdb) info b
Num      Type           Disp Enb Address      What
1        breakpoint      keep y   <PENDING>   cat
2        breakpoint      keep y   0x80100b20  in consoleintr at console.c:309
3        breakpoint      keep y   0x80100678  in printint at console.c:51
(gdb)
```

۲ - برای حذف یک Breakpoint از چه دستوری و چگونه استفاده می شود؟

با استفاده از دستور `delete` و دادن `number` بریک پوینت مورد نظر

```
(gdb) b console.c:50
Breakpoint 3 at 0x80100678: file console.c, line 51.
(gdb) info b
Num      Type           Disp Enb Address      What
1        breakpoint      keep y   <PENDING>   cat
2        breakpoint      keep y   0x80100b20  in consoleintr at console.c:309
3        breakpoint      keep y   0x80100678  in printint at console.c:51
(gdb) delte 3
Undefined command: "delte". Try "help".
(gdb) delete 3
(gdb) info b
Num      Type           Disp Enb Address      What
1        breakpoint      keep y   <PENDING>   cat
2        breakpoint      keep y   0x80100b20  in consoleintr at console.c:309
(gdb) █
```

۳ - دستور `bt` چیست و چگونه کار میکند؟

دستور `bt` برای استفاده از `Backtrace` است .

بک تریس یک `stack trace` نشان میدهد که استکی از تمام فانکشن کال هایی است که تا به این نقطه فراخوانی شده اند.

با `backtrace` میتوان ترتیب فانکشن هایی که کال شده اند و در چه نقطه ای کال شده اند را دید که به دیباگ کمک

میکند . (تصویر در صفحه بعد)

```

312 require(&cons.lock);
(gdb) bt
#0  consoleintr (getc=0x80102f30 <kbdgetc>) at console.c:312
#1  0x80103020 in kbdintr () at kbd.c:51
#2  0x80106335 in trap (tf=0x80116938 <stack+3912>) at trap.c:67
#3  0x8010608f in alltraps () at trapasm.S:20
#4  0x80116938 in stack ()
#5  0x80112cc4 in cpus ()
#6  0x80112cc0 in ?? ()
#7  0x8010387f in mpmain () at main.c:57
#8  0x801039cc in main () at main.c:37
(gdb)

```

۴ - دو تفاوت دستورهای x و print را توضیح دهید. چگونه می توان محتوای یک ثبات خاص را چاپ 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, inspect, p
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.

  -memory-tag-violations [on|off]
    Set printing of memory tag violations for pointers.
    Issue a warning when the printed value is a pointer
    whose logical tag doesn't match the allocation tag of the memory
    location it points to.

  -null-stop [on|off]
    Set printing of char arrays to stop at first null char.

  -object [on|off]
    Set printing of C++ virtual function tables.

  -pretty [on|off]
    Set pretty formatting of structures.

```

دستور X برای این است بتوان محتوای حافظه را با استفاده از آدرس حافظه دید .

x/[n] [f] address

که n برای word هایی که است می‌خواهیم نشان دهد .

که f برای این است که فرمت نمایش محتوای آن آدرس است .

تفاوت های بین print , x :

دستور X برای دیدن محتوای داخل مموری است . در حالی که print برای دیدن مقدار متغیر های برنامه یا هسته است .

دستور X بیشتر برای تست دیتا استراکچر ها و محتوای مموری استفاده میشود . درحالی که print برای مشاهده متغیر های درون برنامه در حال تست .

دستور X میتواند مقدار unit هایی که می‌خواهیم نشان دهیم و همچنین فرمت نمایش انها را دستی تنظیم کنیم .

۵ - برای نمایش وضعیت ثبات‌ها از چه دستوری استفاده می شود؟ متغیرها محلی چطور؟ نتیجه این دستور را در گزارشکار خود بیاورید. همچنین در گزارش خود توضیح دهید که در معماری x86 رجیسترهای edi و esi نشانگر چه چیزی هستند؟

با استفاده دستور info registers میتوان محتوای رجیستر ها (ثبات ها) را دید .

```
(gdb) info registers
eax             0x1             1
ecx             0x0             0
edx             0x0             0
ebx             0x80116938      -2146342600
esp             0x801168e0      0x801168e0 <stack+3824>
ebp             0x801168fc      0x801168fc <stack+3852>
esi             0x80112cc0      -2146358080
edi             0x80112cc4      -2146358076
eip             0x80100b20      0x80100b20 <consoleintr>
eflags          0x86           [ IOPL=0 SF PF ]
cs              0x8            8
ss              0x10           16
ds              0x10           16
es              0x10           16
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             0x3ff000       [ PDBR=1023 PCID=0 ]
```

با استفاده از دستور `info locals` میتوان محتوای متغیرهای محلی را دید .

```
(gdb) info locals
c = <optimized out>
doprocdump = 0
(gdb) █
```

رجیستر `edi (extended Destination Index)` برای مشخص کردن ایندکس مقصد استفاده میشود مثلاً اگر از دستور کپی استفاده کنیم در این رجیستر نوشته میشود که به کدام مقصد در حافظه چیزی کپی شود .
رجیستر `esi(Extended Source Index)` برای مشخص کردن ایندکس مبدا برای وقت هایی که در حال کار با دیتا هستیم کاربرد دارد .
مثلاً برای استفاده از دستوری کپی میگوییم که محتوای حافظه ای که در ثبات `esi` آدرسش ذخیره شده است را بخوان .

۶ - توضیح کلی `struct input` و متغیرهای آن :

```
(gdb) print input
$2 = {buf = "a", '\000' <repeats 126 times>, r = 0, w = 0, e = 1}
(gdb) █
```

این استراکت برای ذخیره بافرهای ورودی است که شامل یک آرایه برای ذخیره بافر ورودی روی ترمینال ، ایندکس اینکه تا کجای بافر خوانده شده . و اینکه تا کجای بافر نوشته شده و اینکه ایندکس آخرین جایی که روی بافر ادیت شده (آخر بافر ورودی که نوشته شده) است .

```
#define INPUT_BUF 128
struct {
    char buf[INPUT_BUF];
    uint r; // Read index
    uint w; // Write index
    uint e; // Edit index
} input;
```

چه زمانی `input.e` تغییر میکند :

هنگامی که ورودی جدیدی از کیبورد به بافر وارد میشود.

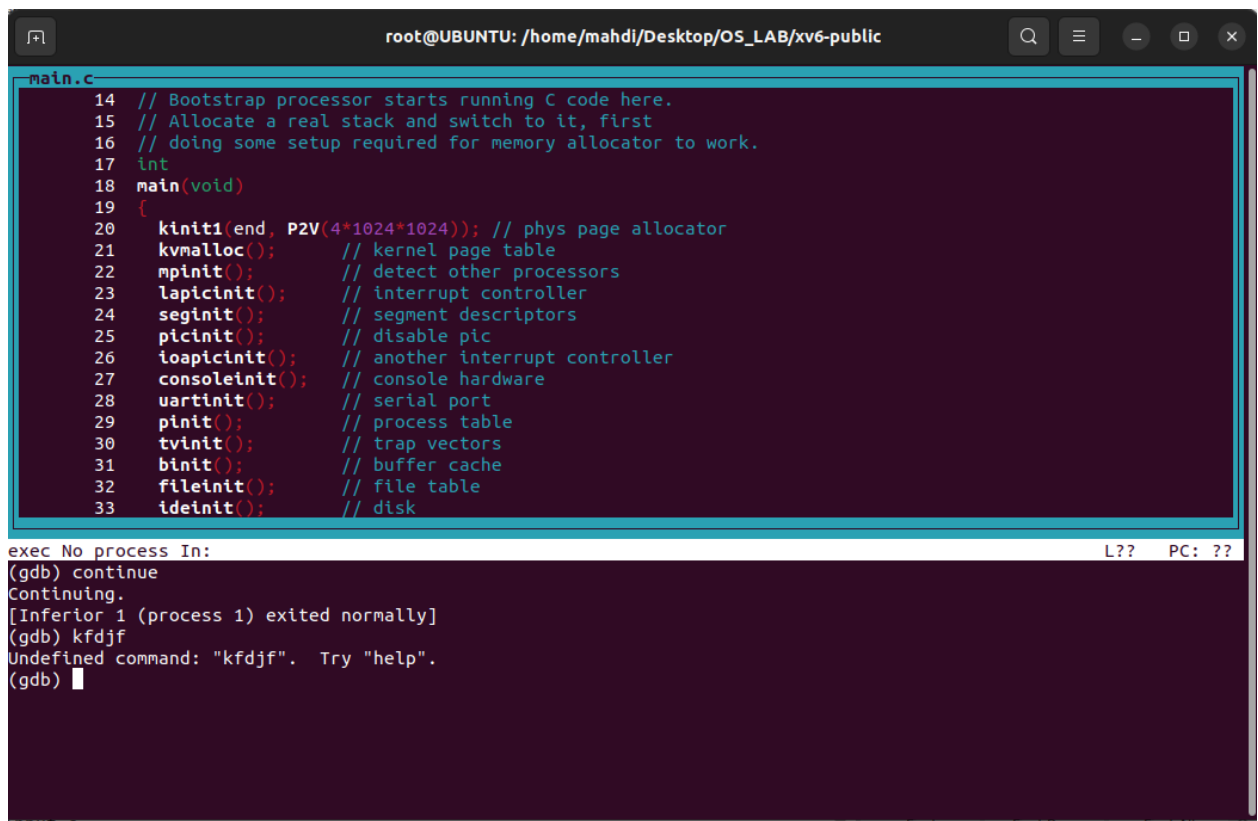
چه زمانی `input.r` تغییر میکند:

زمانی که دستور خواندن از بافر صادر میشود

چه زمانی `input.w` تغییر میکند :

زمانی که کامند تکمیل شده باشد مکان آن جلوتر رفته و با `input.e` یکی میشود.

۷ - TUI چیست و چگونه کار می‌کند؟



```
root@UBUNTU: /home/mahdi/Desktop/OS_LAB/xv6-public
main.c
14 // Bootstrap processor starts running C code here.
15 // Allocate a real stack and switch to it, first
16 // doing some setup required for memory allocator to work.
17 int
18 main(void)
19 {
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
}
exec No process in:
(gdb) continue
Continuing.
[Inferior 1 (process 1) exited normally]
(gdb) kfdjf
Undefined command: "kfdjf". Try "help".
(gdb)
```

با استفاده از TUI میتوان به اشکال زدایی در سطح زبان اسمبلی دست یافت که مزایای خاص خود را دارد که میتوان تغییراتی که کامپایلر بعد از بهینه کردن کد داده را دید و آن را دیباگ کرد .

- دستور `layout src`

این دستور سورس کد C مربوط به آن نقطه ای که توش هستیم رو نمایش میدهد .

- دستور `layout asm`

این دستور سورس کد assembly مربوط به آن نقطه ای که توش هستیم رو نمایش میدهد .

پروژه پیکر بندی و ساختن هسته لینوکس :

در این بخش در آدرس `linux-6.5.8/init/main.c` وارد فایل شده و در تابع `do_initcalls` به شکل فوق به کمک `printk` اسمی اعضای گروه را در آن قرار داده و سیو می کنیم.

```
1297 static void __init do_initcalls(void)
1298 {
1299     int level;
1300     size_t len = saved_command_line_len + 1;
1301     char *command_line;
1302
1303     command_line = kzalloc(len, GFP_KERNEL);
1304     if (!command_line)
1305         panic("%s: Failed to allocate %zu bytes\n", __func__, len);
1306
1307     for (level = 0; level < ARRAY_SIZE(initcall_levels) - 1; level++) {
1308         /* Parser modifies command_line, restore it each time */
1309         strcpy(command_line, saved_command_line);
1310         do_initcall_level(level, command_line);
1311     }
1312     printk("group #20:\nSmahdi Hajiseyedhossein \nAlireza Hosseini \nAmirali Shahriary");
1313
1314     kfree(command_line);
1315 }
```

سپس به کمک نخستین روش پیکربندی هسته یعنی استفاده از تنظیمات پیش فرض دستور `make defconfig` را در پوشه ریشه هسته کد اجرا میکنیم.

در ادامه با کمک دستور `make -j 4 bzImage` میتوانیم کرنل را کامپایل کنیم.

با توجه به دستور کار اکنون با ران کردن دستور `make -j8` که ۸ عملاً همان `ThreadCount` است هسته را ساخت.

در ادامه با کامند زیر امولیتور `qemu` را شروع می کنیم اکنون می توانیم آن را با هسته سفارشی و سیستم فایل اولیه روی 1024 مگابایت رم شروع کنیم.

```
(base) amirali@amirali-ubuntu:~/Downloads/linux-6.5.8/arch/x86/boot$ qemu-system-x86_64 -kernel bzImage -initrd initrd.img-3.6.2 -m 1024
```


پس از ران شدن آن به شکل زیر می شود :

```
QEMU
Machine View
[ 2.853544] Write protecting the kernel read-only data: 26624k
[ 2.855617] Freeing unused kernel image (rodata/data gap) memory: 1616K
[ 2.968379] x86/mm: Checked W+X mappings: passed, no W+X pages found.
[ 2.969141] Run /init as init process
Loading, please wait...
[ 3.182575] input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042
/serio1/input/input3
Starting version 245.4-4ubuntu3.22
[ 5.404599] e1000 0000:00:03:0 enp0s3: renamed from eth0
Begin: Loading essential drivers ... done.
Begin: Running /scripts/init-premount ... done.
Begin: Mounting root file system ... Begin: Running /scripts/local-top ... done.
Begin: Running /scripts/local-premount ... Begin: Waiting for suspend/resume dev
ice ... Begin: Running /scripts/local-block ... done.
done.
Gave up waiting for suspend/resume device
done.
No root device specified. Boot arguments must include a root= parameter.
[ 38.558217] systemd-udevd (74) used greatest stack depth: 12992 bytes left

BusyBox v1.30.1 (Ubuntu 1:1.30.1-4ubuntu6.4) built-in shell (ash)
Enter 'help' for a list of built-in commands.

(initramfs) _
```

با زدن دستور **dmesg** نام اعضای گروه که پیشتر در فایل مشخص کردیم ، قابل رویت خواهد بود.

```
QEMU
Machine View
[ 2.809887] platform regulatory.0: Direct firmware load for regulatory.db fai
led with error -2
[ 2.810254] cfg80211: failed to load regulatory.db
[ 2.811200] ALSA device list:
[ 2.811424]   No soundcards found.
[ 2.811476] group #20:
[ 2.811476] Smahdi Hajiseyedhossein
[ 2.811476] Alireza Hosseini
[ 2.811476] Amirali Shahriary
[ 2.851508] Freeing unused kernel image (initmem) memory: 2608K
[ 2.853544] Write protecting the kernel read-only data: 26624k
[ 2.855617] Freeing unused kernel image (rodata/data gap) memory: 1616K
[ 2.968379] x86/mm: Checked W+X mappings: passed, no W+X pages found.
[ 2.969141] Run /init as init process
[ 2.969193]   with arguments:
[ 2.969206]   /init
[ 2.969218]   with environment:
[ 2.969228]   HOME=/
[ 2.969239]   TERM=linux
[ 3.182575] input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042
/serio1/input/input3
[ 5.404599] e1000 0000:00:03:0 enp0s3: renamed from eth0
[ 38.558217] systemd-udevd (74) used greatest stack depth: 12992 bytes left
[ 100.886303] process '/usr/bin/dmesg' started with executable stack

(initramfs) _
```