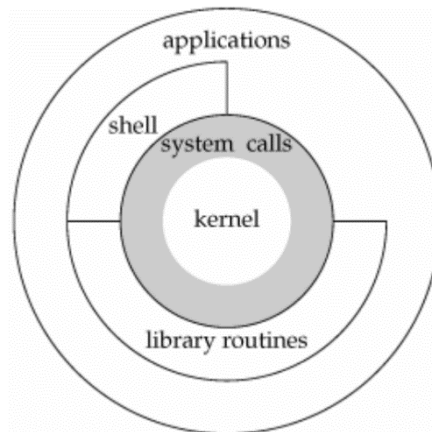


آشنایی با سیستم عامل xv6

1. معماری سیستم عامل xv6

سیستم عامل xv6 مشابه Unix v6 نوشته شده و معماری و ساختاری شبیه به آن دارد. این سیستم عامل برای پردازنده‌های مبتنی بر x86 نوشته شده (مطابق با داکيومنت این سیستم عامل؛ xv6-rev11). علاوه بر آن در دفاع از این سخن می‌توان به فایل x86.h اشاره کرد که از دستورات پردازنده‌های x86 استفاده شده است. در دیگر فایل‌های "basic headers"، نظیر asm.h و mmu.h نیز می‌توان اشاراتی به معماری x86 مشاهده کرد. معماری کلی سیستم عامل Unix بصورت زیر می‌باشد:



همانطور که گفته شد، معماری xv6 نیز از Unix پیروی می‌کند. این موضوع از دسته‌بندی فایل‌ها که شامل user-level, system calls, file systems و... می‌شود نیز قابل مشاهده است.

2. بخش‌های پردازش و چگونگی اختصاص پردازنده به پردازنده‌های مختلف

یک پردازش در xv6 از حافظه فضای کاربری (user-space) (شامل دستورات، داده‌ها و استک)، و وضعیت پردازش که فقط برای هسته قابل رؤیت است تشکیل شده است.

xv6 زمان را بین پردازنده‌ها تقسیم می‌کند و به صورت نامحسوس پردازنده‌ها را برای اجرای دستورات به پردازنده‌ها اختصاص می‌دهد. هر وقت یک پردازش قرار است از اجرا توسط پردازنده خارج شود، سیستم عامل register های CPU که حاوی مقادیر مورد نیاز آن پردازش بوده را ذخیره می‌کند تا دفعه بعدی که آن پردازش قرار است اجرا شود، آنها را بازگرداند.

هسته xv6 به هر پردازش یک شناسه یکتا (PID (Process Identifier) اختصاص می‌دهد. با استفاده از system call `getpid()` می‌توان PID پردازش کنونی را دریافت کرد.

3. مفهوم file descriptor و عملکرد pipe در xv6

مفهوم file descriptor در این سیستم عامل‌ها در واقع به یک عدد اشاره می‌کند که به کمک آن می‌تواند از یک فایل بخواند یا در آن بنویسد. به ازای هر پردازش یک جدول برای نگهداری file descriptor ها وجود دارد که باعث می‌شود این مقادیر برای پردازنده‌ها به صورت خصوصی باشد و برای هر کدام از آنها از مقدار 0 آغاز شود.

طبق قرارداد، مقدار 0 برای stdin، مقدار 1 برای stdout و مقدار 2 برای stderr تعریف شده است. با توجه به اینکه file descriptor می‌تواند مربوط به یک فایل، یک دستگاه یا pipe باشد، سیستم عامل با استفاده از پیاده‌سازی file descriptorها به این شکل، توانسته است یک interface انتزاعی برای هرکدام از این موارد ایجاد کند و همه آن‌ها را به یک شکل ببیند.

عملگر pipe برای ارتباط بین پردازنده‌ها استفاده می‌شود. در واقع به کمک این عملگر می‌توانیم stdout یک پردازنده را به stdin یک پردازنده دیگر متصل کنیم.

عملکرد pipe در سیستم عامل xv6 به این صورت است که ابتدا به کمک تابع pipe، دو file descriptor به هم متصل هستند ایجاد می‌کند. سپس برای پردازنده سمت چپ ابتدا بخش قابل خواندن پایپ را می‌بندد و سپس بخش قابل نوشتن آن را به عنوان stdout برای این پردازنده قرار می‌دهد و دستور را اجرا می‌کند. برای پردازنده سمت راست ابتدا بخش قابل نوشتن پایپ را می‌بندد و سپس بخش قابل خواندن آن را به عنوان stdin در نظر می‌گیرد و در نهایت این دستور را هم اجرا می‌کند. سپس منتظر می‌ماند تا هر 2 دستور خاتمه یابند. ممکن است دستور سمت راست پایپ شامل دستوراتی باشد که در خود آن‌ها نیز از پایپ استفاده شده است. در این صورت، درختی از دستورات اجرا می‌شوند. لازم به ذکر است که پردازنده سمت راست تا زمانی که stdin آن به end of file نرسیده باشد، منتظر داده جدید می‌ماند.

4. توابع exec و fork

تابع fork برای ایجاد یک process جدید استفاده می‌شود. در واقع این تابع یک نسخه کپی از پردازنده‌ای می‌سازد که این تابع را صدا زده است. منظور از کپی این است که دیتا و دستورات پردازنده فعلی در حافظه پردازنده جدید (child) کپی می‌شوند. با وجود اینکه در لحظه ایجاد پردازنده فرزند، داده‌های آن (متغیرها و رجیسترها) با پردازنده پدر یکسان هستند، اما در واقع این دو پردازنده حافظه جداگانه‌ای خواهند داشت و تغییر یک متغیر در پردازنده پدر، آن متغیر در پردازنده فرزند را تغییر نمی‌دهد. پردازنده پدر پس از ایجاد پردازنده فرزند، به caller تابع fork باز می‌گردد که امکان اجرای همزمان دو پردازنده را فراهم می‌سازد. مقدار return شده از تابع fork نیز pid پردازنده فرزند خواهد بود. نقطه شروع پردازنده فرزند نیز دقیقاً همان caller تابع fork است با این تفاوت که مقدار خروجی این تابع عدد 0 خواهد بود. پس اگر با استفاده از `pid = fork();` یک پردازنده جدید درست کنیم، یکی از حالت‌های زیر برای مقدار pid رخ می‌دهد:

- `pid = 0`: در پردازنده فرزند هستیم.
 - `pid > 0`: در پردازنده پدر هستیم و مقدار pid در واقع آی‌دی پردازنده فرزند است.
 - `pid < 0`: در زمان اجرای تابع fork و پردازنده جدید اروری وجود داشته و پردازنده فرزند ایجاد نشده است.
- اگر پس از fork کردن از تابع `wait(int*)` استفاده شود، پردازنده پدر منتظر پایان یافتن پردازنده فرزند می‌شود و سپس کار خود را ادامه می‌دهد. خروجی این تابع، pid پردازنده پایان یافته است. اگر پردازنده فعلی هیچ پردازنده فرزندی نداشته باشد، خروجی این تابع 1- خواهد بود. همچنین ورودی این تابع پوینتر به متغیری است که در نهایت status code مربوط به پردازنده فرزند در آن قرار می‌گیرد. برای ignore کردن این پارامتر از `((int*)0)` استفاده می‌شود.

قطعه کد زیر مثالی برای استفاده از تابع fork را نشان می‌دهد:

```
int pid = fork();
if (pid == 0) {
    printf("This is child process\n");
    printf("Child process is exiting\n");
    exit(0);
}
else if (pid > 0) {
    printf("This is parent process\n");
    printf("Waiting for child process to exit\n");
    wait((int*)0);
    printf("Child process exited\n");
}
else {
    printf("Fork failed\n");
}
}
```

تابع `exec` حافظه پردازش فعلی را با یک حافظه جدید که در آن یک برنامه با فایل ELF لود شده است، جایگزین می‌کند. در واقع `exec()` راهی برای اجرای یک برنامه در پردازش فعلی است. بر خلاف تابع `fork()`، برنامه به caller تابع `exec()` باز نمی‌گردد و برنامه جدید اجرا می‌شود مگر اینکه در زمان اجرای این تابع یک ارور رخ دهد. برنامه جدید اجرا شده در یک نقطه‌ای با استفاده از تابع `exit` اجرای پردازش را خاتمه می‌دهد. تابع `exec` دو پارامتر ورودی دارد که پارامتر اول نام فایل برنامه و پارامتر دوم آرایه آرگومان‌های ورودی برنامه است. قطعه کد زیر مثالی از اجرای این تابع را نشان می‌دهد:

```
char* args[] = {"ls", "-l", "/home", NULL}; // Null is required
exec("/bin/ls", args);
printf("Exec failed\n");
```

مزیت ادغام نکردن این دو تابع در زمان I/O redirection خودش را نشان می‌دهد. زمانی که کاربر در shell یک برنامه را اجرا می‌کند، کاری که در پشت صحنه توسط shell انجام می‌شود به شرح زیر است:

1. ابتدا دستور تایپ شده توسط کاربر در ترمینال را می‌خواند.
 2. با استفاده از تابع `fork` یک پردازش جدید ایجاد می‌کند.
 3. در پردازش فرزند با استفاده از تابع `exec` برنامه درخواست شده توسط کاربر را جایگزین پردازش فعلی (فرزند) می‌کند.
 4. در پردازش پدر برای اتمام کار پردازش فرزند `wait` می‌کند.
 5. پس از اتمام پردازش فرزند به `main` باز می‌گردد و منتظر دستور جدید می‌شود.
- زمانی که کاربر برای یک دستور از redirection استفاده می‌کند، تغییرات لازم در file descriptor ها پس از `fork` و پیش از `exec` و در پردازش فرزند انجام می‌شود.
- قطعه کد زیر این مورد را به شکل ساده شده نشان می‌دهد (فرض کنید دستور اجرا شده `cat < in.txt` است):

```
char* args = {"cat", NULL};
int pid = fork();
if (pid == 0) {
    close(0); // close stdin
    open("in.txt", O_RDONLY); // open in.txt for reading (fd: 0)
    exec("/bin/cat", args);
    printf("Exec failed\n");
}
else if (pid > 0) {
    wait((int *)0);
    printf("Child process has exited\n");
}
else {
    printf("The fork failed\n");
}
```

در صورتی که این دو تابع ادغام شوند، یا باید حالت‌های redirection به عنوان پارامتر به تابع `forkexec` پاس داده شوند که هندل کردن این حالت در دسرهای خودش را دارد و یا اینکه shell پیش از اجرای این تابع، file descriptor خود را تغییر دهد و بعد از اتمام کار این تابع نیز به حالت قبل برگرداند و یا در بدترین حالت، هندل کردن redirection را در هر برنامه مانند cat پیاده‌سازی کنیم.

اضافه کردن یک متن به Boot Message

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
Group 1:
- Saman Eslami Nazari : 810199375
- Pasha Barahimi : 810199385
- Misagh Mohaghegh : 810199484
$ |
```

پس از بوت شدن سیستم عامل نام اعضای گروه نمایش داده شده است.
این کار با افزودن یک printf در فایل init.c انجام شده است.

اضافه کردن چند قابلیت به کنسول xv6

1. دستور ctrl + n برای پاک کردن همه اعداد خط کنونی

برای اضافه کردن این قابلیت تابع زیر به فایل console.c اضافه شده است و در switch case تابع consoleintr به ازای حالت C('N')، این تابع صدا زده می شود:

```
static void
remnums()
{
    char cmd[INPUT_BUF];
    int j = 0;
    for(int i = 0; i < input.e - input.w; ++i){
        int idx = (input.w + i) % INPUT_BUF;
        if(input.buf[idx] >= '0' && input.buf[idx] <= '9'){
            continue;
        }
        cmd[j++] = input.buf[idx];
    }
    cmd[j] = '\0';
    consclear();
    consputs(cmd);
}
```

2. دستور ctrl + r برای برعکس کردن خط کنونی

همانند قابلیت قبلی، با صدا کردن تابع زیر در switch case تابع consoleintr به ازای حالت C('R')، می توان این قابلیت را به ترمینال اضافه کرد:

```
static void
revline()
{
    char cmd[INPUT_BUF];
    memmove(cmd, input.buf + input.w, input.e - input.w);
    cmd[input.e - input.w] = '\0';
    revstr(cmd, input.e - input.w);
    consclear();
    consputs(cmd);
}
```

3. دستور tab برای تکمیل کردن خط کنونی

استراکت زیر به منظور ذخیره تاریخچه دستورات استفاده می‌شود:

```
#define HIST_SIZE 15
struct {
    uint queue_idx;
    char cmd_buf[HIST_SIZE][INPUT_BUF];
    uint last_used_idx;

    int is_suggestion_used;
    char original_cmd[INPUT_BUF];
    uint original_cmd_size;
} hist;
```

متغیر `queue_idx` نشان‌دهنده شماره خانه فعلی صف تاریخچه دستورات است. `cmd_buf` نیز آرایه نگه‌دارنده دستورات گذشته است. علاوه بر آن از یک شماره خانه به نام `last_used_idx` برای نگه داشتن آخرین پیشنهاد فعلی استفاده می‌کنیم تا هنگام پیشنهاد دادن دستورات مختلف، همواره دستوری جدید از تاریخچه به کاربر ارائه شود. سه متغیر بعدی برای نگه داشتن دستور اصلی است که کاربر قبل از دریافت پیشنهادات وارد کرده بود؛ استفاده از این متغیرها به این دلیل است که پیشنهادات همواره بر مبنای دستور وارد شده باشند و پس از تغییر دستور توسط پیشنهادات، پیشنهادات عوض نشوند.

دو تابع `get_suggestion` و `suggest_cmd` نیز به فایل `console.c` اضافه شده‌اند که به ترتیب وظایف پیشنهاد مناسب برای دستور فعلی و قرار دادن پیشنهاد دریافت شده در کنسول را بر عهده دارند. علاوه بر آن تابع `push_current_hist` نیز برای اضافه کردن دستور وارد شده توسط کاربر به تاریخچه دستورات نوشته شده است. این توابع به صورت زیر می‌باشند:

```

static int
get_suggestion(const char* cmd, uint cmd_size)
{
    for(int i = 0; i < HIST_SIZE; ++i){
        int idx = (i + hist.last_used_idx) % HIST_SIZE;
        if(strncmp(cmd, hist.cmd_buf[idx], cmd_size) == 0){
            return idx;
        }
    }
    return -1;
}

static void
suggest_cmd()
{
    if(!hist.is_suggestion_used){
        hist.original_cmd_size = input.e - input.w;
        memmove(hist.original_cmd, input.buf + input.w,
hist.original_cmd_size);
    }
    int suggested_cmd = get_suggestion(hist.original_cmd,
hist.original_cmd_size);
    if(suggested_cmd >= 0){
        hist.is_suggestion_used = 1;
        hist.last_used_idx = suggested_cmd + 1;
        consclear();
        consputs(hist.cmd_buf[suggested_cmd]);
    }
}

static void
push_current_hist()
{
    memset(hist.cmd_buf[hist.queue_idx], 0, INPUT_BUF);
    memmove(hist.cmd_buf[hist.queue_idx],
        input.buf + input.w,
        input.e - input.w - 1);
    hist.queue_idx = (hist.queue_idx + 1) % HIST_SIZE;
    hist.is_suggestion_used = 0;
    hist.last_used_idx = 0;
    memset(hist.original_cmd, 0, INPUT_BUF);
}

```

در نهایت عملکرد پیاده‌سازی شده در این بخش بدین شرح می‌باشد: به ازای هر دستور وارد شده توسط کاربر، آن دستور در تاریخچه دستورات قرار می‌گیرد. این تاریخچه شامل حداکثر 15 دستور آخر می‌باشد. علاوه بر آن دستورات قرار گرفته در این تاریخچه صحت‌سنجی نشده و دستورات نامعتبری نیز ذخیره می‌شوند. هر بار که کاربر کلید tab را فشار دهد، یک پیشنهاد توسط این تاریخچه به کاربر ارائه می‌شود که تکمیل کننده رشته وارد شده است. پیشنهادات از اولین دستورات وارد شده شروع می‌شوند و با هر بار فشردن مجدد tab دستور بعدی

پیشنهاد داده می‌شود. در صورت اتمام پیشنهادات و پیمایش تمام دستورات موجود در تاریخچه، پیشنهادات دوباره از اول ارائه خواهند شد و اینکار بصورت دایره‌وار ادامه خواهد یافت.

اجرا و پیاده‌سازی یک برنامه سطح کاربر

```
$ prime_numbers 20 50
$ cat prime_numbers.txt
23 29 31 37 41 43 47
$ |
```

فایل prime_numbers.c همانند برنامه های دیگر از جمله wc.c و mkdir.c نوشته شده و به متغیر UPROGS در Makefile اضافه شده است.

مقدمه‌ای درباره سیستم عامل و xv6

5. سه وظیفه اصلی سیستم عامل

6. گروه‌های فایل های اصلی xv6

کامپایل سیستم عامل xv6

7. دستور make -n و کدام دستور فایل نهایی را می‌سازد؟

8. متغیرهای UPROGS و ULIB در Makefile

متغیر UPROGS: این متغیر لیستی از برنامه‌های کاربر را دارد که در هنگام ساخت و کامپایل xv6، این برنامه‌ها نیز کامپایل و تبدیل به فایل‌های قابل اجرا توسط سیستم عامل می‌شوند. نام هر یک از این برنامه‌ها به صورت `_file_name` در این لیست قرار گرفته است. تمام اسامی به صورت `_file_name` (اسامی که یک - ابتدایشان دارند)، یک هدف¹ با پیشنیاز²های فایل آجکت هدف (`file_name.o`) و متغیر ULIB دارد. بنابراین هدف‌های موجود در UPROGS منجر به ساخت فایل آجکت برنامه‌های کاربر، اجرا شدن هدف‌های مربوط به ULIB می‌شود و در نهایت اجرای دستور `ld` می‌شود. دستور `ld` برای پیوند³ فایل‌های مورد نیاز و تولید یک فایل قابل اجرا مورد استفاده قرار می‌گیرد. علاوه بر آن فایل‌های آجکت مربوط به هر برنامه (`file_name.o`) توسط یک قانون درونی⁴ Makefile ساخته می‌شوند و به صورت صریح در Makefile نوشته نشده‌اند.

متغیر ULIB: این متغیر شامل تعدادی از کتابخانه‌های زبان C می‌باشد. در بسیاری از کدهای xv6 توابع این کتابخانه‌ها استفاده شده‌اند و برای اجرایشان به کامپایل این فایل‌ها نیاز داریم. برای مثال برنامه‌های سطح کاربر نیازمند کامپایل فایل‌های ULIB می‌باشند؛ بنابراین همانطور که در بخش قبل نیز گفته شد، فایل‌های ULIB به عنوان پیشنیاز در قوانین قرار گرفته‌اند و در نهایت توسط دستور `ld` به فایل‌های اجرایی پیوند می‌شوند. فایل‌های ULIB شامل توابعی مانند `printf`، `strcpy`، `strcmp`، `malloc` و... هستند.

در نهایت، همانطور که از اسم این متغیرها نیز پیداست، UPROGS معادل User Programs و ULIB معادل User Libraries است که به ترتیب برنامه‌های کاربر و کتابخانه‌های کاربر محسوب می‌شوند.

اجرا بر روی شبیه‌ساز QEMU

9. محتوای دو دیسک ورودی QEMU

مراحل بوت سیستم عامل xv6

اجرای بوت‌لودر

¹ Target

² Prerequisite

³ Link

⁴ Built-in implicit rule

10. محتوای سکتور نخست دیسک قابل بوت

اولین کامندهای اجرا شونده توسط Makefile شامل کامپایل کردن object file های bootmain.c و bootasm.S، پیوند زدن این دو و تولید bootblock.o، objcopy کردن بخش text. فایل bootblock.o به فایل bootblock و در نهایت داده شدن به اسکریپت sign.pl برای اضافه کردن 2 بایت boot signature به bootblock است.

در سکتور نخست (512 بایت اول) دیسک قابل بوت، محتوای فایل bootblock قرار دارد.

11. مقایسه فایل باینری بوت با بقیه فایل‌های باینری xv6 و تبدیل آن به اسمبلی

همه فایل‌های باینری آبجکت xv6 در فرمت ELF¹ هستند. این فرمت باینری از بخش‌های مختلفی تشکیل شده است. در ابتدای آن هدرهایی شامل اطلاعات لود شدن فایل نوشته شده است و سپس چند section دارد که هر کدام حجمی از کد یا داده اند که در آدرس مشخصی از حافظه لود می‌شوند. فرمت فایل ELF برای انواع object file ها یعنی relocatable (فایل‌های .o) که توسط linker استفاده می‌شوند، executable و shared object ها تعریف شده است.

دو هدر ELF Header و Program Header در فایل elf.h به زبان سی تعریف شده‌اند.

در ELF Header بخشی به نام e_entry وجود دارد که آدرس نقطه ورود برنامه را مشخص می‌کند.

از section های ELF می‌توان به text. و .rodata و .data و .bss. اشاره کرد.

- **text**: شامل دستورات قابل اجرای برنامه است.
- **.rodata**: حاوی داده‌های read-only از جمله string literal ها در زبان سی است.
- **.data**: شامل داده‌های مقدار دهی شده مانند برخی متغیرهای گلوبال است.
- **.bss**: شامل داده‌های مقدار دهی نشده است که چون داده‌ای وجود ندارد فقط آدرس و اندازه اش در فایل ذخیره می‌شود.

با استفاده از دستور `objdump -h bootblock.o` می‌توانیم نوع فایل باینری (که مانند بقیه فایل‌های باینری xv6 به فرمت elf32-i386 است)، و در ادامه خروجی دستور، section های ELF را مشاهده کنیم.

بوت لودر پس از لود شدن در آدرس ثابت 0x7C00، توسط پردازنده اجرا می‌شود تا کرنل را اجرا کند. در اینجا تنها اطلاعات مهم، کدی است که قرار است اجرا بشود. با مقایسه bootblock.o با بقیه object file ها می‌بینیم که بخش‌های data. و غیره را ندارد و بخش اصلی اش فقط text. است.

از آنجا که bootblock.o در آدرس خاصی شروع به اجرا شدن می‌کند، در هنگام ساخته شدنش از فلگ Ttext 0x7C00 - استفاده شده است که آدرس بخش text. فایل خروجی را مشخص می‌کند. فلگ -e start هم می‌گوید که نقطه شروع برنامه لیبل start در bootasm.S است.

خود فایلی که در سکتور بوت قرار دارد یعنی bootblock با استفاده از دستور:

```
objcopy -S -O binary -j .text bootblock.o bootblock
```

(و اضافه کردن boot signature) تولید می‌شود. این فلگ‌های objcopy در **بخش بعدی** توضیح داده شده‌اند. این دستور محتویات بخش text. را به صورت raw binary به فایل bootblock می‌ریزد. این یعنی فایل bootblock از فرمت ELF پیروی نمی‌کند و هیچ هدری هم ندارد. این فایل با دیگر فایل‌های باینری xv6 تفاوت دارد و کد قابل اجرای خالص بدون هیچ اطلاعات اضافه‌ای است.

¹ Executable and Linkable Format

پس نوع فایل دودویی مربوط به بوت raw binary است (که در حالت کلی چیز مشخصی نیست) و اینجا همان محتویات بخش text. (instruction های قابل اجرا بر روی معماری x86) می باشد.

این با بقیه فایل های باینری از آنجا که به فرمت ELF نیست تفاوت دارد.

دلیل استفاده نکردن از ELF برای bootblock این است که فرمت ELF را هسته سیستم عامل می داند و نه CPU. پس وقتی که هسته هنوز اجرا نشده نمی توان فرمت ELF را خواند. اگر BIOS فایل bootblock.o را برای بوت شدن به CPU می داد، از آنجا که CPU هدرهای ELF را نمی شناسد همه محتوای فایل را به دید instruction ها نگاه کرده و برداشت اشتباهی از آن می کند. پس باید فقط دستورات خالص را به CPU داد. یک دلیل دیگر هم کم کردن حجم فایل است. با استخراج بخش text. فایل bootblock.o، حجم آن کاهش یافته و در 510 بایت جا می گیرد.

برای تبدیل bootblock به اسمبلی، از کامند زیر استفاده می کنیم:

```
objdump -D -b binary -m i386 -M addr16,data16 bootblock
```

از آنجا که bootblock باینری خام است و هیچ هدری برای مشخص کردن معماری اش ندارد، آنها را باید دستی به objdump بدهیم. فلگ هایی که استفاده شده:

- -D : برای disassemble کردن باینری.
- -b binary : نوع فایل را raw binary در نظر می گیریم.
- -m i386 : معماری اسمبلی فایل را مشخص می کنیم.
- -M addr16,data16 : از آنجا که وقتی BIOS سکتور بوت را لود می کند در real mode هستیم و

CPU در حالت 16 بیت است، اسمبلی 16 بیت نیز استفاده شده است پس هنگام disassemble

کردن هم می گوئیم که آدرس ها و داده ها را 16 بیت در نظر بگیرد.

می توانیم با استفاده از فلگ -adjust-vma=0x7C00 آدرس شروع قرار گرفتن اسمبلی خروجی در حافظه را تغییر بدهیم که مانند واقعیت از آدرس 0x7C00 شروع بشود.

با مشاهده خروجی کامند می بینیم که ابتدای آن بسیار مشابه با bootasm.S است:

```
bootblock:      file format binary

Disassembly of section .data:

00007c00 <.data>:
7c00:      fa                cli
7c01:      31 c0               xor     %ax,%ax
7c03:      8e d8               mov     %ax,%ds
7c05:      8e c0               mov     %ax,%es
7c07:      8e d0               mov     %ax,%ss
7c09:      e4 64               in      $0x64,%al
7c0b:      a8 02               test    $0x2,%al
7c0d:      75 fa               jne     0x7c09
7c0f:      b0 d1               mov     $0xd1,%al
7c11:      e6 64               out     %al,$0x64
7c13:      e4 64               in      $0x64,%al
7c15:      a8 02               test    $0x2,%al
7c17:      75 fa               jne     0x7c13
7c19:      b0 df               mov     $0xdf,%al
7c1b:      e6 60               out     %al,$0x60
7c1d:      0f 01 16 78 7c      lgdtw   0x7c78
7c22:      0f 20 c0             mov     %cr0,%eax
7c25:      66 83 c8 01         or      $0x1,%eax
7c29:      0f 22 c0             mov     %eax,%cr0
7c2c:      ea 31 7c 08 00      ljmp    $0x8,$0x7c31
7c31:      66 b8 10 00 8e d8   mov     $0xd8e0010,%eax
7c37:      8e c0               mov     %ax,%es
7c39:      8e d0               mov     %ax,%ss
7c3b:      66 b8 00 00 8e e0   mov     $0xe08e0000,%eax
7c41:      8e e8               mov     %ax,%gs
7c43:      bc 00 7c             mov     $0x7c00,%sp
7c46:      00 00               add     %al,(%bx,%si)
7c48:      e8 f0 00             call    0x7d3b
```

12. علت استفاده از objcopy در هنگام make

با استفاده از این دستور می‌توان محتویات یک فایل object را در یک فایل object دیگر کپی کرد. برای این کار نیازی نیست فرمت فایل ورودی با فرمت فایل مقصد یکسان باشد. با توجه به اینکه این برنامه کار ترجمه فایل را با استفاده از کتابخانه BFD انجام می‌دهد، تمامی فرمت‌های موجود در این کتابخانه پشتیبانی می‌شوند و امکان تبدیل بین آن‌ها وجود دارد. این دستور برای ترجمه فایل‌های object از فایل‌های موقت (temp) استفاده می‌کند و سپس آن‌ها را پاک می‌کند. آپشن‌هایی از این دستور که در Makefile مربوط به xv6 استفاده شده‌اند به طور خلاصه در بخش زیر توضیح داده شده است:

- **S-:** در صورت استفاده از این آپشن، اطلاعات مربوط به symbol table و relocation records در فایل مقصد حذف می‌شوند. داده‌های symbol table نام و مکان متغیرها و فرآیندهایی را ذخیره می‌کنند که ممکن است در فایل‌های object دیگر از آن‌ها استفاده شده باشد. داده‌های relocation records نیز اطلاعاتی در مورد آدرس‌هایی از فایل object ذخیره می‌کند که در هنگام ساخت فایل مشخص نبوده و نیاز است در ادامه توسط linker مقداردهی شوند. این آدرس‌ها می‌توانند مربوط به متغیرها و توابعی باشند که در فایل‌های دیگر تعریف شده‌اند و در خود فایل وجود ندارند. در این حالت linker در زمان لینک کردن فایل‌ها، این آدرس‌ها را مقداردهی می‌کند.
- **0-:** این آپشن نوع فرمت فایل مقصد را نشان می‌دهد. برای مثال با استفاده از آپشن **binary 0-** فایل تولید شده از نوع raw binary خواهد بود. این نوع فایل‌ها به فرمت خاصی نوشته نشده‌اند. از جمله این فایل‌ها می‌توان به فایل‌های memory dump اشاره کرد.
- **-j:** با استفاده از این آپشن می‌توانیم تنها بخشی از فایل object را به فایل جدید کپی کنیم. در این Makefile در چند بخش زیر از دستور objcopy استفاده شده است:
 1. در bootblock پس از لینک شدن bootmain.o و bootasm.o در فایلی به نام bootblock.o، محتویات بخش text. این فایل را در یک فایل raw binary به نام bootblock کپی می‌کند. سپس این فایل را به اسکریپت sign.pl می‌دهد که ابتدا سایز فایل را بررسی می‌کند که بیشتر از 510 بایت نباشد و سپس 2 بایت 0x55 و 0xAA که boot signature اند را به انتهای فایل اضافه می‌کند.
 2. در entryother محتویات بخش text. فایل bootblockother.o را در یک فایل raw binary به نام entryother کپی می‌کند.
 3. در initcode محتویات فایل initcode.out در یک فایل raw binary به نام initcode کپی می‌شود. در نهایت با لینک شدن فایل‌های entry.o و فایل‌های object که در متغیر OBJS تعریف شده‌اند و فایل‌های باینری initcode و entryother که پیش‌تر با استفاده از دستور objcopy ساخته شدند، فایل kernel ساخته می‌شود.

13. چرا برای بوت کردن فقط از فایل C استفاده نشده و اسمبلی هم هست؟

چون که برخی از کارها نیازمند دسترسی سطح پایین به سیستم می‌باشند و با کد C نمی‌توان آن‌ها را انجام داد. یک نمونه از این کارها وارد شدن به protected mode است. وقتی که BIOS کد سکتور بوت را لود می‌کند، پردازنده x86 در real mode اجرا می‌شود. در این حالت آدرس دهی حافظه همیشه فیزیکی است، پردازنده 16 بیت است و فقط 1 مگابایت حافظه داریم. برای اینکه بتوانیم از پردازنده 32 بیت استفاده کنیم و تا 4 گیگابایت حافظه داشته باشیم، باید وارد protected mode بشویم که این کار فقط در اسمبلی (با 1 کردن بیت اول Control Register 0) ممکن است.

14. وظیفه ثبات‌های x86

- ثبات عام منظوره: پردازنده‌های x86 دارای 8 ثبات عام منظوره هستند. از این ثبات‌ها می‌توان به ثبات انباشت‌کننده¹ اشاره کرد که یک ثبات میانی برای ذخیره خروجی بخش محاسباتی (ALU) است. نام این ثبات از این رو انباشت‌کننده نهاده شده که پس از هر بار انجام محاسبات، نتیجه در آن ذخیره شده و در محاسبات بعدی از مقدار ذخیره شده در آن به عنوان ورودی استفاده می‌شود و دوباره نتیجه آن در همین ثبات ذخیره می‌شود. به عبارتی دیگر بصورت نوبتی، نتایج محاسبات در آن انباشت می‌شوند.
- ثبات قطعه: پردازنده‌های x86 دارای 6 ثبات قطعه هستند. یک ثبات قطعه، ثبات پشته² می‌باشد. این ثبات اطلاعاتی مربوط به قطعه‌ای از حافظه را ذخیره می‌کند که از آن برای پشته فراخوانی³ استفاده می‌شود. دقت شود که ثبات قطعه پشته (SS) با ثبات نشانگر پشته (SP) تفاوت دارد؛ برای اطلاعات بیشتر در این خصوص به [این پیوست](#) مراجعه کنید.
- ثبات وضعیت: ثبات FLAGS، ثبات وضعیتی است که نشان‌دهنده حالت فعلی پردازنده است. این ثبات مخصوص پردازنده‌های 16 بیتی است. EFLAGS و RFLAGS ثبات‌های مشابه برای پردازنده‌های 32 بیتی و 64 بیتی می‌باشند. هر بیت از این ثبات نشان‌دهنده یک پرچم⁴ برای یک وضعیت می‌باشد که می‌تواند حالت درست یا غلط داشته باشد. این پرچم‌ها نشان‌دهنده وضعیت اعمال منطقی و محاسباتی یا محدودیت‌های اعمال شده بر عملیات فعلی پردازنده هستند. واضح است که عملکرد این پرچم‌ها به تعداد بیت‌های رجیستر و معماری پردازنده بستگی دارد. ثبات FLAGS برای پردازنده Intel x86 به شرح زیر می‌باشد:

Intel x86 FLAGS register			
بیت	مخفف	توضیح	دسته‌بندی
0	CF	Carry flag	وضعیت
1		رزرو شده	
2	PF	Parity flag	وضعیت
3		رزرو شده	
4	AF	Adjust Flag	وضعیت
5		رزرو شده	
6	ZF	Zero flag	وضعیت
7	SF	Sign flag	وضعیت
8	TF	Trap flag	کنترل
9	IF	Interrupt enable flag	کنترل

¹ Accumulator register (AX)² Stack³ Call stack⁴ Flag

کنترل	Direction flag	DF	10
وضعیت	Overflow flag	OF	11
سیستم	سطح دسترسی ورودی خروجی	IOPL	12-13
سیستم	پرچم فعالیت تو در تو	NT	14
	رزرو شده		15

- ثبات کنترلی: این نوع از ثبات‌ها مسئول تغییر در رفتار کلی پردازنده و یا دیگر دستگاه‌های مرتبط اند. از این دسته ثبات‌ها می‌توان به ثبات CR0 اشاره کرد که در پردازنده‌های 32بیتی مانند i386 و بالاتر استفاده می‌شود. بیت‌های این ثبات نشان‌دهنده تغییرات و کنترل‌های مختلفی در رفتار کلی پردازنده هستند که به شرح زیر می‌باشد:

نام	مخفف	بیت
Protected Mode Enable	PE	0
Monitor co-processor	MP	1
Emulation	EM	2
Task switched	TS	3
Extension type	ET	4
Numeric error	NE	5
Write protect	WP	16
Alignment mask	AM	18
Not-write through	NW	29
Cache disabled	CD	30
Paging	PG	31

15. نقص اصلی real mode پردازنده x86

16. آدرس‌دهی حافظه در real mode

17. کد bootmain.c چرا هسته را در آدرس 0x100000 قرار می‌دهد؟

18. کد معادل entry.s در هسته لینوکس

کد معادل entry.S برای معماری x86 در هسته لینوکس:

<https://github.com/torvalds/linux/blob/master/arch/x86/entry/entry.S>

که برای 32بیت و 64بیت جدا است.

اجرای هسته xv6

19. دلیل فیزیکی بودن آدرس page table

برای تبدیل آدرس مجازی به آدرس فیزیکی نیازمند جدول ذکر شده هستیم و برای دسترسی به این جدول نیاز به آدرس آن داریم. در صورتی که آدرس این جدول به صورت مجازی ذخیره شود، برای پیدا کردن آدرس فیزیکی‌اش به خودش نیاز خواهیم داشت و حلقه بی‌نهایتی به وجود می‌آید که این حالت باعث ایجاد تناقض می‌شود و هیچ وقت نمی‌توانیم به این جدول دسترسی پیدا کنیم. در صورتی که بخواهیم از یک جدول دیگر برای پیدا کردن آدرس فیزیکی این جدول استفاده کنیم، در نهایت نیاز به یک آدرس فیزیکی برای پایان دادن به حلقه خواهیم داشت. در نتیجه آدرس دسترسی به این جدول به صورت فیزیکی ذخیره می‌شود.

20. توابع entry.s را توضیح دهید و تابع معادل در هسته لینوکس را بیابید

21. مختصری راجع به محتوای فضای آدرس مجازی هسته

22. چرا برای کد و داده های سطح کاربر پرچم SEG_USER تنظیم شده است؟

قطعه‌بندی در x86 در تابع `segininit` و در تکه کد زیر انجام می‌شود:

```
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);
```

علاوه بر آن تعریف SEG به صورت زیر می‌باشد:

```
#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 }
```

بنابراین همانطور که مشخص است (و در توضیحات آزمایش نیز آمده است) تمام قطعه‌های هسته و کاربر یک بخش از حافظه را در اختیار دارند. هر یک از این قطعه‌ها با یک دسکریپتور در GDT^1 مشخص شده که این دسکریپتور شامل اطلاعاتی مانند آدرس شروع قطعه، اندازه قطعه و سطح دسترسی قطعه می‌باشد. برای خواندن یک دستورالعمل، ابتدا قطعه آن از طریق دسکریپتورش یافت می‌شود (که در اینجا قطعه کد دسکریپتور هسته و کاربر یکسان اند) و سپس صفحه مربوط به آن پس از طی مراحل مربوطه پیدا می‌شود. پس از این مراحل و تبدیل آدرس منطقی به آدرس فیزیکی، دستورالعمل از حافظه خوانده شده و اجرا می‌شود. موضوعی که در این مرحله باید به آن دقت کرد، سطح دسترسی مورد نیاز یک دستور برای اجرای آن است. هنگامی که مکان قطعه از روی دسکریپتور قطعه مشخص می‌شود، سطح دسترسی فعلی یا همان CPL^2 نیز از روی سطح دسترسی دسکریپتور یا همان DPL^3 مشخص می‌شود. بدین گونه از طریق DPL متفاوت می‌توان سطح دسترسی فعلی دستورالعمل‌ها را نیز تعیین کرد؛ حتی اگر این دسکریپتورها قطعات یکسانی از حافظه را تعریف کنند.

برای مثال دستورالعمل `IN`، وظیفه خواندن یک بایت از پورت را دارد و این عمل نیازمند این است که سطح دسترسی فعلی مقداری ممتازتر از سطح دسترسی ورودی/خروجی داشته باشد (سطح دسترسی ورودی/خروجی در رجیستر وضعیت `FLAG` مشخص شده است) که این مقدار در لینوکس برابر صفر است؛ مقدار دسترسی

¹ Global descriptor table

² Current privilege level

³ Descriptor privilege level

فعلی (CPL) برابر مقدار سطح دسترسی دسکریپتور (DPL) قطعه‌ای است که کد مربوط به این دستور العمل در آن قرار گرفته است و اگر این دستورات عمل در قطعه کاربر قرار گرفته باشد قابل اجرا نخواهد بود؛ چرا که قطعه مربوط به کد کاربر، سطح دسترسیش برابر DPL_USER یا همان 3 (کمترین میزان دسترسی) است. بنابراین با وجود اینکه هر دو بخش کاربر و هسته به قطعات یکسانی دسترسی دارند، اما سطح دسترسی متفاوتی داشته و کاربر هر دستورات عملی را نمی‌تواند اجرا کند.

اجرای نخستین برنامه سطح کاربر

23. اجزای struct proc و معادل آن در لینوکس

این struct که برای ذخیره وضعیت هر پردازش به کار می‌رود، در فایل proc.h تعریف شده و 13 متغیر در آن قرار دارد:

- `uint sz` : حجم و اندازه حافظه گرفته شده توسط پردازش به واحد بایت.
- `pde_t* pgdir` : پوینتر به page table پردازش است. (pde: page directory entry)
- `char* kstack` : پوینتر به kernel stack است. استک کرنل قسمتی از kernel space است و نه user space و برای اجرای syscalls ها از برنامه استفاده می‌شود.
- `enum procstate state` : این enum وضعیت پردازش را مشخص می‌کند و می‌تواند به حالت‌های procstate یعنی UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE باشد.
- `int pid` : این عدد PID (Processor Identifier) است که عدد یکتایی بین همه پردازش‌ها است.
- `struct proc* parent` : پوینتر به پردازش پدر (پردازش سازنده پردازش کنونی توسط تابع fork) است. تایپ این پوینتر مثل خود پردازش کنونی struct proc است.
- `struct trapframe* tf` : پوینتر به trap frame برای ذخیره وضعیت اجرای برنامه در هنگام اجرای یک syscall.
- `struct context* context` : پوینتر به struct context است که مقادیر رجیسترهای مورد نیاز برای context switching را نگه می‌دارد. با استفاده از تابع switch (که با اسمبلی تعریف شده) می‌توان به یک پردازش switch کرد.
- `void* chan` : در صورتی که مقدار آن 0 نباشد، یعنی پردازش خوابیده است (برای کاری wait می‌کند). اینجا chan به معنای channel است و چنل‌های متعددی از جمله چنل خط ورود کنسول داریم.
- `int killed` : در صورتی که مقدار آن 0 نباشد یعنی پردازش kill شده است.
- `struct file* ofile[NOFILE]` : آرایه‌ای از پوینترها به فایل‌های باز شده توسط پردازش است.
- `struct inode* cwd` : این متغیر current working directory را مشخص می‌کند.
- `char name[16]` : نام پردازش برای اشکال زدایی.

معادل این struct در هسته لینوکس:

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

استراکت task_struct در فایل <linux/sched.h>

24. چرا به خواب رفتن در کد مدیریت‌کننده سیستم مشکل‌ساز است؟

25. تفاوت فضای آدرس هسته با فضای آدرس توسط kvmalloc

26. تفاوت فضای آدرس inituvم با فضای آدرس کاربر در کد مدیریت سیستم

27. کدام بخش از آماده سازی سیستم بین تمامی هسته های پردازنده مشترک و کدام بخش

اختصاصی است؟

هسته اول که فرآیند بوت را انجام می دهد توسط کد entry.S وارد تابع main در فایل main.c می شود. تمامی توابع آماده سازی سیستم که در این تابع فراخوانده شده اند توسط این هسته اجرا می شوند. از طرفی، هسته های دیگر از طریق کد entryother.S وارد تابع mpenter می شوند. در این تابع نیز 4 تابع برای آماده سازی فراخوانده می شوند. در نتیجه می توان گفت این 4 تابع بین تمامی هسته ها مشترک خواهند بود. یکی از این توابع به نام switchkvm به صورت مستقیم با هسته اول مشترک نیست. این تابع در mpenter صدا زده می شود در صورتی که در تابع main وجود ندارد. در واقع تابع kvmalloc که در main صدا زده می شود به صورت زیر است:

```
void
kvmalloc(void)
{
    kpgdir = setupkvm();
    switchkvm();
}
```

خط اول تابع یک page table برای کرنل ایجاد می کند که این مورد توسط هسته اول انجام می پذیرد. پس از آن باید هسته به این page table سوییچ کند که این کار در تمامی هسته ها انجام می پذیرد. بخش هایی از آماده سازی سیستم که در تمام هسته ها مشترک هستند به شرح زیر است:

- switchkvm
- seginit
- lapicinit
- mpmain

همچنین بخش هایی که تنها در هسته اول (به صورت اختصاصی) اجرا می شوند به شرح زیر است:

- kinit1
- 0setupkvm00kvmalloc
- mpinit
- picinit
- ioapicinit
- consoleinit
- uartinit
- pinit
- tvinit
- binit
- fileinit
- ideinit
- startothers
- kinit2
- userinit

از موارد اختصاصی هسته اول می توان به تابع startothers اشاره کرد که واضح است فقط پردازنده اول نیاز است بقیه پردازنده ها را start کند و نیازی نیست هر پردازنده در زمان بالا آمدن این کار را انجام دهد. یا

برای مثال زمانی که پردازنده اول به کمک تابع `ideinit` دیسک را شناسایی می‌کند، نیازی نیست بقیه پردازنده‌ها این کار را انجام دهند.

از طرفی، همه پردازنده‌ها باید آدرس `page table` که توسط پردازنده اول ایجاد شده را در رجیستر خود ذخیره کنند در نتیجه تابع `switchkvm` بین همه آن‌ها مشترک است. همچنین، همه پردازنده‌ها باید کار خود را شروع کنند و آماده اجرای برنامه‌ها شوند که این مورد توسط تابع `mpmain` انجام می‌پذیرد. در نتیجه این تابع هم بین تمام پردازنده‌ها مشترک خواهد بود.

زمان‌بند که توسط تابع `scheduler` انجام می‌پذیرد در تابع `mpmain` صدا زده می‌شود که این تابع بین تمامی هسته‌ها مشترک است. این مورد از کامنت‌های داکيومنت تابع ذکر شده نیز قابل برداشت است:

```
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
```

هر پردازنده `scheduler` مربوط به خودش را خواهد داشت و در نتیجه این تابع بین تمامی پردازنده‌ها مشترک است.

28. برنامه معادل `initcode.s` در هسته لینوکس

اشکال زدایی

روند اجرای GDB

1. دستور مشاهده breakpointها

برای مشاهده breakpointهای فعلی می‌توان از دستور `info breakpoints` استفاده کرد:

```
(gdb) b cat.c:12
Breakpoint 1 at 0x93: file cat.c, line 12.
(gdb) info breakpoints
Num      Type             Disp Enb Address          What
1        breakpoint      keep y   0x00000093 in cat at cat.c:12
```

2. دستور حذف یک breakpoint

برای حذف یک breakpoint می‌توان از دستور `del <breakpoint_number>` استفاده کرد. مقدار `breakpoint number` را از طریق دستور `info break` می‌توان مشاهده کرد. برای مثال در نمونه زیر، دستور `info breakpoint` دو breakpoint در خطوط 12 و 14 فایل `cat.c` را نشان می‌دهد:

```
(gdb) info break
Num      Type             Disp Enb Address          What
1        breakpoint      keep y   0x00000097 in cat at cat.c:12
2        breakpoint      keep y   0x000000dc in cat at cat.c:14
```

حال با استفاده از دستور `del 2`، breakpoint واقع شده در خط 14 را حذف کرده و سپس مجدداً با دستور `info break` صحت این عمل را تایید می‌کنیم:

```
(gdb) del 2
(gdb) info break
Num      Type           Disp Enb Address      What
1        breakpoint    keep y   0x00000097 in cat at cat.c:12
```

بنابراین با استفاده از دستور `delete` یا همان `del` و با بهره‌گیری از شماره breakpoint می‌توان آن را حذف کرد. علاوه بر آن با استفاده از دستور `clear` و مکان مشخص breakpoint (ترکیبی از نام فایل و شماره خط آن به صورت `<file_name>:<line_number>`) می‌توان breakpoint موجود در آن مکان را پاک کرد.

کنترل روند اجرا و دسترسی به حالت سیستم

3. خروجی bt

دستور `bt` که مخفف `backtrace` است `call stack` برنامه در لحظه کنونی (در حین متوقف بودن روند اجرای برنامه) را نشان می‌دهد.

هر تابع که صدا زده می‌شود یک `stack frame` مخصوص به خودش را می‌گیرد که متغیرهای محلی و آدرس بازگشت و غیره در آن قرار دارند.

خروجی این دستور در هر خط یک `stack frame` را نشان می‌دهد که به ترتیب از درونی‌ترین `frame` که در آن قرار داریم شروع می‌شود.

می‌توان با دستور `bt n` که `n` یک عدد است فقط `n` فریم درونی‌تر را نشان داد و با دستور `bt -n` فقط `n` فریم بیرونی‌تر را نشان داد.

برای استفاده از این دستور می‌توان از کلیدواژه‌های مختلفی استفاده کرد از جمله:

`bt`, `backtrace`, `where`, `info stack`

در مثال زیر، در خط 15 فایل `wc.c` یک breakpoint گذاشته شده است. این خط کد، داخل تابعی به نام `wc` قرار دارد که از داخل تابع `main` ورودی برنامه `wc` صدا می‌شود.

پس از اجرای کامند `README wc` مشاهده می‌کنیم که روی خط 15 متوقف شده و دستور `bt` به طور صحیح `call stack` را نشان می‌دهد.

```
Reading symbols from _wc...
(gdb) break 15
Breakpoint 1 at 0xa0: file wc.c, line 15.
(gdb) continue
Continuing.
[ 1b: a0] 0x250 <strchr>: push %ebp

Thread 1 hit Breakpoint 1, wc (fd=3, name=0x2ff4 "README") at wc.c:15
15      while((n = read(fd, buf, sizeof(buf))) > 0){
(gdb) bt
#0  wc (fd=3, name=0x2ff4 "README") at wc.c:15
#1  0x00000056 in main (argc=2, argv=0x2fe8) at wc.c:50
(gdb) |
```

4. تفاوت دستور x و print

همانطور که در `help` این دو دستور نوشته شده است، با استفاده از دستور `print` (به اختصار `p`) می‌توان مقدار یک متغیر را چاپ کرد. آرگومان ورودی این دستور، نام متغیر خواهد بود.

با استفاده از دستور `x` می‌توان محتویات یک خانه حافظه را چاپ کرد. بدیهی‌ست که آرگومان ورودی این دستور، آدرس خانه حافظه مذکور است.

لازم به ذکر است که هر دو دستور ذکر شده می‌توانند فرمت خروجی را به صورت `FMT` / در آرگومان‌های ورودی خود دریافت کنند.

در مثال زیر پس از دستور `cat prime_numbers.txt` متغیر `fd` چاپ می‌شود. برای پیدا کردن آدرس این متغیر نیز از دستور `print &fd` استفاده شده است:

```
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, cat (fd=3) at cat.c:12
12      while((n = read(fd, buf, sizeof(buf))) > 0) {
(gdb) print fd
$1 = 3
(gdb) print &fd
$2 = (int *) 0x2f90
(gdb) x 0x2f90
0x2f90: 0x00000003
(gdb) x/d 0x2f90
0x2f90: 3
```

همچنین برای مشاهده مقدار یک ثابت خاص می‌توان از دستور `info registers <reg_name>` استفاده کرد:

```
(gdb) info registers eax
eax          0x3          3
```

5. نمایش وضعیت ثبات‌ها و متغیرهای محلی؛ رجیسترهای `esi` و `edi`

با استفاده از دستور `info register` می‌توان وضعیت ثبات‌ها را مشاهده کرد. علاوه بر آن از مخفف این دستور یعنی `i r` نیز می‌توان استفاده کرد. خروجی این دستور به شرح زیر می‌باشد:

```
(gdb) info registers
eax            0x0                0
ecx            0x0                0
edx            0x0                0
ebx            0x82                130
esp            0x8010b500          0x8010b500 <stack+3904>
ebp            0x8010b508          0x8010b508 <stack+3912>
esi            0x80113540          -2146355904
edi            0x80112fa4          -2146357340
eip            0x80103bf5          0x80103bf5 <mycpu+21>
eflags         0x46                [ IOPL=0 ZF 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=0 PCID=0 ]
cr4            0x10               [ PSE ]
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}}
--Type <RET> for more, q to quit, c to continue without paging--
```

برای مشاهده متغیرهای محلی نیز می‌توان از دستور `info locals` استفاده کرد. خروجی این دستور پس برای اشکال‌زدایی فایل `cat.c` به صورت زیر می‌باشد:

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

ثبات SI مخفف Source Index بوده و برای اشاره به یک مبدا در عملیات stream به کار می‌رود. DI نیز مخفف Destination Index بوده و برای اشاره به یک مقصد در عملیات stream به کار می‌رود. E در ابتدای اسامی این ثبات‌ها به معنی Extended بوده و در حالت 32 بیت به کار می‌رود. SI به عنوان نشانگر داده و به عنوان مبدا در برخی عملیات مربوط به رشته‌ها استفاده می‌شود. DI نیز به عنوان نشانگر داده و مقصد برخی عملیات مربوط به رشته‌ها استفاده می‌شود.

6. ساختار `struct input`

این `struct` در فایل `console.c` تعریف شده است و برای خط ورودی کنسول سیستم عامل استفاده می‌شود. این استراکت در کد چنین تعریف شده است:

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

یعنی از یک instance به نام input از یک unnamed struct استفاده می‌شود. این را در GDB هم می‌توان با کامند `ptype` برای پرینت کردن تایپ یک متغیر مشاهده کرد:

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

آرایه `buf` بافر و محل ذخیره خط ورودی است که اندازه آن حداکثر 128 کاراکتر است.

متغیرهای دیگر عدد هستند و هر کدام ایندکس‌های را برای `buf` را مشخص می‌کنند.

- متغیر `w` محل شروع نوشتن خط ورودی کنونی در `buf` است.
- متغیر `e` محل کنونی کرسر در خط ورودی است.
- متغیر `r` برای خواندن `buf` استفاده می‌شود. (از `w` قبلی شروع می‌کند)

نحوه تغییر این متغیرها را با یک مثال می‌بینیم:

در ابتدای کار مقادیر اولیه متغیرها را پرینت می‌کنیم و یک breakpoint در تابع `consoleintr` در انتهای بخش default، (جایی که اینتر یا `ctrl+d` زده می‌شود یا کرسر از `buf` فراتر می‌رود) می‌گذاریم:

```
(gdb) print input
$1 = {buf = '\000' <repeats 127 times>, r = 0, w = 0, e = 0}
(gdb) break console.c:340
Breakpoint 1 at 0x80100dc6: file console.c, line 340.
```

حال `continue` می‌کنیم و عبارت `test` را وارد می‌کنیم:

```
(gdb) print input
$2 = {buf = "test\n", '\000' <repeats 122 times>, r = 0, w = 5, e = 5}
```

طبق breakpoint ای که گذاشتیم اجرای برنامه متوقف می‌شود.

می‌بینیم که ورودی در `buf` قرار گرفته و متغیر `e` به 5 تغییر یافته که مکان بعد از آخرین حرف `buf` است.

حال دوباره `continue` کرده و دستی (با `ctrl+c`) روند اجرا را متوقف می‌کنیم:

```
(gdb) print input
$2 = {buf = "test\n", '\000' <repeats 122 times>, r = 5, w = 5, e = 5}
```

می‌بینیم که مقدار `r` به همان مقدار `w` رسیده است. یعنی از `w` قبلی (که 0 بود) شروع کرده و به `w` کنونی می‌رسد تا کل خط را بخواند. (با گذاشتن یک watchpoint می‌توان دقیق‌تر بررسی کرد که `r` یکی یکی جلو می‌رود)

این بار عبارت `another` را وارد می‌کنیم:

```
(gdb) print input
$3 = {buf = "test\nanother\n", '\000' <repeats 114 times>, r = 5, w = 13, e = 13}
```

`w` باز هم به آخر `buf` رفته و `e` هم در ابتدای خط ورودی جدید است پس با `w` برابر است.

اگر برنامه را `continue` و سپس متوقف کنیم، می‌بینیم که `r` به `w` می‌رسد:

```
(gdb) print input
$3 = {buf = "test\nanother\n", '\000' <repeats 114 times>, r = 13, w = 13, e = 13}
```


حال `continue` کرده و عبارت `xyz` را می‌نویسیم ولی اینتر نمی‌زنیم و دستی برنامه را متوقف می‌کنیم:

```
(gdb) print input
$4 = {buf = "test\nanother\nxyz", '\000' <repeats 111 times>, r = 13, w = 13, e = 16}
```

طبق انتظار متغیر `e` جلو رفته است. اگر کاراکتر آخر را پاک کنیم:

```
(gdb) print input
$5 = {buf = "test\nanother\nxyz", '\000' <repeats 111 times>, r = 13, w = 13, e = 15}
```

`e` یک واحد به عقب بر می‌گردد.

توجه که هر حرکت کرسر خود 3 کاراکتر در این بافر می‌ریزد و مقدار `e` را افزایش دهد حتی اگر رو به عقب باشد.

اشکال زدایی در سطح کد اسمبلی

7. خروجی دستورهای `layout src` و `layout asm` در TUI

در محیط TUI با استفاده از دستور `layout src` می‌توان کد سورس برنامه در حال دیباگ را نمایش داد:

```
cat.c
4
5 char buf[512];
6
7 void
8 cat(int fd)
9 {
10     int n;
11
12     while((n = read(fd, buf, sizeof(buf))) > 0) {
13         if (write(1, buf, n) != n) {
14             printf(1, "cat: write error\n");
15             exit();
16         }
17     }
18     if(n < 0){
19         printf(1, "cat: read error\n");
```

remote Thread 1.1 In: cat L12 PC: 0x93
(gdb) layout src

همچنین با استفاده از دستور `layout asm` می‌توانیم کد اسمبلی برنامه در حال دیباگ را مشاهده کنیم:

```
0xf0 <cat+96>    push    %eax
0xf1 <cat+97>    push    %eax
0xf2 <cat+98>    push    $0x7fa
0xf7 <cat+103>   push    $0x1
0xf9 <cat+105>   call    0x4c0 <printf>
0xfe <cat+110>  call    0x363 <exit>
0x103           xchg    %ax,%ax
0x105           xchg    %ax,%ax
0x107           xchg    %ax,%ax
0x109           xchg    %ax,%ax
0x10b           xchg    %ax,%ax
0x10d           xchg    %ax,%ax
0x10f           nop
0x110 <strcpy>   push    %ebp
0x111 <strcpy+1> xor     %eax,%eax
0x113 <strcpy+3> mov     %esp,%ebp
```

remote Thread 1.1 In: cat L12 PC: 0x93
(gdb) layout src
(gdb) layout asm

در نهایت با استفاده از دستور `layout split` می‌توانیم کد سورس برنامه و اسمبلی آن را به طور همزمان مشاهده کنیم:

```

cat.c
10  int n;
11
12  while((n = read(fd, buf, sizeof(buf))) > 0) {
13      if (write(1, buf, n) != n) {
14          printf(1, "cat: write error\n");
15          exit();
16      }
17  }
18
19  }
20
21  }
22
23  }
24
25  }
26
27  }
28
29  }
30
31  }
32
33  }
34
35  }
36
37  }
38
39  }
40
41  }
42
43  }
44
45  }
46
47  }
48
49  }
50
51  }
52
53  }
54
55  }
56
57  }
58
59  }
60
61  }
62
63  }
64
65  }
66
67  }
68
69  }
70
71  }
72
73  }
74
75  }
76
77  }
78
79  }
80
81  }
82
83  }
84
85  }
86
87  }
88
89  }
90
91  }
92
93  }
94
95  }
96
97  }
98
99  }
100
101  }
102
103  }
104
105  }
106
107  }
108
109  }
110
111  }
112
113  }
114
115  }
116
117  }
118
119  }
120
121  }
122
123  }
124
125  }
126
127  }
128
129  }
130
131  }
132
133  }
134
135  }
136
137  }
138
139  }
140
141  }
142
143  }
144
145  }
146
147  }
148
149  }
150
151  }
152
153  }
154
155  }
156
157  }
158
159  }
160
161  }
162
163  }
164
165  }
166
167  }
168
169  }
170
171  }
172
173  }
174
175  }
176
177  }
178
179  }
180
181  }
182
183  }
184
185  }
186
187  }
188
189  }
190
191  }
192
193  }
194
195  }
196
197  }
198
199  }
200
201  }
202
203  }
204
205  }
206
207  }
208
209  }
210
211  }
212
213  }
214
215  }
216
217  }
218
219  }
220
221  }
222
223  }
224
225  }
226
227  }
228
229  }
230
231  }
232
233  }
234
235  }
236
237  }
238
239  }
240
241  }
242
243  }
244
245  }
246
247  }
248
249  }
250
251  }
252
253  }
254
255  }
256
257  }
258
259  }
260
261  }
262
263  }
264
265  }
266
267  }
268
269  }
270
271  }
272
273  }
274
275  }
276
277  }
278
279  }
280
281  }
282
283  }
284
285  }
286
287  }
288
289  }
290
291  }
292
293  }
294
295  }
296
297  }
298
299  }
300
301  }
302
303  }
304
305  }
306
307  }
308
309  }
310
311  }
312
313  }
314
315  }
316
317  }
318
319  }
320
321  }
322
323  }
324
325  }
326
327  }
328
329  }
330
331  }
332
333  }
334
335  }
336
337  }
338
339  }
340
341  }
342
343  }
344
345  }
346
347  }
348
349  }
350
351  }
352
353  }
354
355  }
356
357  }
358
359  }
360
361  }
362
363  }
364
365  }
366
367  }
368
369  }
370
371  }
372
373  }
374
375  }
376
377  }
378
379  }
380
381  }
382
383  }
384
385  }
386
387  }
388
389  }
390
391  }
392
393  }
394
395  }
396
397  }
398
399  }
400
401  }
402
403  }
404
405  }
406
407  }
408
409  }
410
411  }
412
413  }
414
415  }
416
417  }
418
419  }
420
421  }
422
423  }
424
425  }
426
427  }
428
429  }
430
431  }
432
433  }
434
435  }
436
437  }
438
439  }
440
441  }
442
443  }
444
445  }
446
447  }
448
449  }
450
451  }
452
453  }
454
455  }
456
457  }
458
459  }
460
461  }
462
463  }
464
465  }
466
467  }
468
469  }
470
471  }
472
473  }
474
475  }
476
477  }
478
479  }
480
481  }
482
483  }
484
485  }
486
487  }
488
489  }
490
491  }
492
493  }
494
495  }
496
497  }
498
499  }
500
501  }
502
503  }
504
505  }
506
507  }
508
509  }
510
511  }
512
513  }
514
515  }
516
517  }
518
519  }
520
521  }
522
523  }
524
525  }
526
527  }
528
529  }
530
531  }
532
533  }
534
535  }
536
537  }
538
539  }
540
541  }
542
543  }
544
545  }
546
547  }
548
549  }
550
551  }
552
553  }
554
555  }
556
557  }
558
559  }
560
561  }
562
563  }
564
565  }
566
567  }
568
569  }
570
571  }
572
573  }
574
575  }
576
577  }
578
579  }
580
581  }
582
583  }
584
585  }
586
587  }
588
589  }
590
591  }
592
593  }
594
595  }
596
597  }
598
599  }
600
601  }
602
603  }
604
605  }
606
607  }
608
609  }
610
611  }
612
613  }
614
615  }
616
617  }
618
619  }
620
621  }
622
623  }
624
625  }
626
627  }
628
629  }
630
631  }
632
633  }
634
635  }
636
637  }
638
639  }
640
641  }
642
643  }
644
645  }
646
647  }
648
649  }
650
651  }
652
653  }
654
655  }
656
657  }
658
659  }
660
661  }
662
663  }
664
665  }
666
667  }
668
669  }
670
671  }
672
673  }
674
675  }
676
677  }
678
679  }
680
681  }
682
683  }
684
685  }
686
687  }
688
689  }
690
691  }
692
693  }
694
695  }
696
697  }
698
699  }
700
701  }
702
703  }
704
705  }
706
707  }
708
709  }
710
711  }
712
713  }
714
715  }
716
717  }
718
719  }
720
721  }
722
723  }
724
725  }
726
727  }
728
729  }
730
731  }
732
733  }
734
735  }
736
737  }
738
739  }
740
741  }
742
743  }
744
745  }
746
747  }
748
749  }
750
751  }
752
753  }
754
755  }
756
757  }
758
759  }
760
761  }
762
763  }
764
765  }
766
767  }
768
769  }
770
771  }
772
773  }
774
775  }
776
777  }
778
779  }
780
781  }
782
783  }
784
785  }
786
787  }
788
789  }
790
791  }
792
793  }
794
795  }
796
797  }
798
799  }
800
801  }
802
803  }
804
805  }
806
807  }
808
809  }
810
811  }
812
813  }
814
815  }
816
817  }
818
819  }
820
821  }
822
823  }
824
825  }
826
827  }
828
829  }
830
831  }
832
833  }
834
835  }
836
837  }
838
839  }
840
841  }
842
843  }
844
845  }
846
847  }
848
849  }
850
851  }
852
853  }
854
855  }
856
857  }
858
859  }
860
861  }
862
863  }
864
865  }
866
867  }
868
869  }
870
871  }
872
873  }
874
875  }
876
877  }
878
879  }
880
881  }
882
883  }
884
885  }
886
887  }
888
889  }
890
891  }
892
893  }
894
895  }
896
897  }
898
899  }
900
901  }
902
903  }
904
905  }
906
907  }
908
909  }
910
911  }
912
913  }
914
915  }
916
917  }
918
919  }
920
921  }
922
923  }
924
925  }
926
927  }
928
929  }
930
931  }
932
933  }
934
935  }
936
937  }
938
939  }
940
941  }
942
943  }
944
945  }
946
947  }
948
949  }
950
951  }
952
953  }
954
955  }
956
957  }
958
959  }
960
961  }
962
963  }
964
965  }
966
967  }
968
969  }
970
971  }
972
973  }
974
975  }
976
977  }
978
979  }
980
981  }
982
983  }
984
985  }
986
987  }
988
989  }
990
991  }
992
993  }
994
995  }
996
997  }
998
999  }
1000
1001  }
1002
1003  }
1004
1005  }
1006
1007  }
1008
1009  }
1010
1011  }
1012
1013  }
1014
1015  }
1016
1017  }
1018
1019  }
1020
1021  }
1022
1023  }
1024
1025  }
1026
1027  }
1028
1029  }
1030
1031  }
1032
1033  }
1034
1035  }
1036
1037  }
1038
1039  }
1040
1041  }
1042
1043  }
1044
1045  }
1046
1047  }
1048
1049  }
1050
1051  }
1052
1053  }
1054
1055  }
1056
1057  }
1058
1059  }
1060
1061  }
1062
1063  }
1064
1065  }
1066
1067  }
1068
1069  }
1070
1071  }
1072
1073  }
1074
1075  }
1076
1077  }
1078
1079  }
1080
1081  }
1082
1083  }
1084
1085  }
1086
1087  }
1088
1089  }
1090
1091  }
1092
1093  }
1094
1095  }
1096
1097  }
1098
1099  }
1100
1101  }
1102
1103  }
1104
1105  }
1106
1107  }
1108
1109  }
1110
1111  }
1112
1113  }
1114
1115  }
1116
1117  }
1118
1119  }
1120
1121  }
1122
1123  }
1124
1125  }
1126
1127  }
1128
1129  }
1130
1131  }
1132
1133  }
1134
1135  }
1136
1137  }
1138
1139  }
1140
1141  }
1142
1143  }
1144
1145  }
1146
1147  }
1148
1149  }
1150
1151  }
1152
1153  }
1154
1155  }
1156
1157  }
1158
1159  }
1160
1161  }
1162
1163  }
1164
1165  }
1166
1167  }
1168
1169  }
1170
1171  }
1172
1173  }
1174
1175  }
1176
1177  }
1178
1179  }
1180
1181  }
1182
1183  }
1184
1185  }
1186
1187  }
1188
1189  }
1190
1191  }
1192
1193  }
1194
1195  }
1196
1197  }
1198
1199  }
1200
1201  }
1202
1203  }
1204
1205  }
1206
1207  }
1208
1209  }
1210
1211  }
1212
1213  }
1214
1215  }
1216
1217  }
1218
1219  }
1220
1221  }
1222
1223  }
1224
1225  }
1226
1227  }
1228
1229  }
1230
1231  }
1232
1233  }
1234
1235  }
1236
1237  }
1238
1239  }
1240
1241  }
1242
1243  }
1244
1245  }
1246
1247  }
1248
1249  }
1250
1251  }
1252
1253  }
1254
1255  }
1256
1257  }
1258
1259  }
1260
1261  }
1262
1263  }
1264
1265  }
1266
1267  }
1268
1269  }
1270
1271  }
1272
1273  }
1274
1275  }
1276
1277  }
1278
1279  }
1280
1281  }
1282
1283  }
1284
1285  }
1286
1287  }
1288
1289  }
1290
1291  }
1292
1293  }
1294
1295  }
1296
1297  }
1298
1299  }
1300
1301  }
1302
1303  }
1304
1305  }
1306
1307  }
1308
1309  }
1310
1311  }
1312
1313  }
1314
1315  }
1316
1317  }
1318
1319  }
1320
1321  }
1322
1323  }
1324
1325  }
1326
1327  }
1328
1329  }
1330
1331  }
1332
1333  }
1334
1335  }
1336
1337  }
1338
1339  }
1340
1341  }
1342
1343  }
1344
1345  }
1346
1347  }
1348
1349  }
1350
1351  }
1352
1353  }
1354
1355  }
1356
1357  }
1358
1359  }
1360
1361  }
1362
1363  }
1364
1365  }
1366
1367  }
1368
1369  }
1370
1371  }
1372
1373  }
1374
1375  }
1376
1377  }
1378
1379  }
1380
1381  }
1382
1383  }
1384
1385  }
1386
1387  }
1388
1389  }
1390
1391  }
1392
1393  }
1394
1395  }
1396
1397  }
1398
1399  }
1400
1401  }
1402
1403  }
1404
1405  }
1406
1407  }
1408
1409  }
1410
1411  }
1412
1413  }
1414
1415  }
1416
1417  }
1418
1419  }
1420
1421  }
1422
1423  }
1424
1425  }
1426
1427  }
1428
1429  }
1430
1431  }
1432
1433  }
1434
1435  }
1436
1437  }
1438
1439  }
1440
1441  }
1442
1443  }
1444
1445  }
1446
1447  }
1448
1449  }
1450
1451  }
1452
1453  }
1454
1455  }
1456
1457  }
1458
1459  }
1460
1461  }
1462
1463  }
1464
1465  }
1466
1467  }
1468
1469  }
1470
1471  }
1472
1473  }
1474
1475  }
1476
1477  }
1478
1479  }
1480
1481  }
1482
1483  }
1484
1485  }
1486
1487  }
1488
1489  }
1490
1491  }
1492
1493  }
1494
1495  }
1496
1497  }
1498
1499  }
1500
1501  }
1502
1503  }
1504
1505  }
1506
1507  }
1508
1509  }
1510
1511  }
1512
1513  }
1514
1515  }
1516
1517  }
1518
1519  }
1520
1521  }
1522
1523  }
1524
1525  }
1526
1527  }
1528
1529  }
1530
1531  }
1532
1533  }
1534
1535  }
1536
1537  }
1538
1539  }
1540
1541  }
1542
1543  }
1544
1545  }
1546
1547  }
1548
1549  }
1550
1551  }
1552
1553  }
1554
1555  }
1556
1557  }
1558
1559  }
1560
1561  }
1562
1563  }
1564
1565  }
1566
1567  }
1568
1569  }
1570
1571  }
1572
1573  }
1574
1575  }
1576
1577  }
1578
1579  }
1580
1581  }
1582
1583  }
1584
1585  }
1586
1587  }
1588
1589  }
1590
1591  }
1592
1593  }
1594
1595  }
1596
1597  }
1598
1599  }
1600
1601  }
1602
1603  }
1604
1605  }
1606
1607  }
1608
1609  }
1610
1611  }
1612
1613  }
1614
1615  }
1616
1617  }
1618
1619  }
1620
1621  }
1622
1623  }
1624
1625  }
1626
1627  }
1628
1629  }
1630
1631  }
1632
1633  }
1634
1635  }
1636
1637  }
1638
1639  }
1640
1641  }
1642
1643  }
1644
1645  }
1646
1647  }
1648
1649  }
1650
1651  }
1652
1653  }
1654
1655  }
1656
1657  }
1658
1659  }
1660
1661  }
1662
1663  }
1664
1665  }
1666
1667  }
1668
1669  }
1670
1671  }
1672
1673  }
1674
1675  }
1676
1677  }
1678
1679  }
1680
1681  }
1682
1683  }
1684
1685  }
1686
1687  }
1688
1689  }
1690
1691  }
1692
1693  }
1694
1695  }
1696
1697  }
1698
1699  }
1700
1701  }
1702
1703  }
1704
1705  }
1706
1707  }
1708
1709  }
1710
1711  }
1712
1713  }
1714
1715  }
1716
1717  }
1718
1719  }
1720
1721  }
1722
1723  }
1724
1725  }
1726
1727  }
1728
1729  }
1730
1731  }
1732
1733  }
1734
1735  }
1736
1737  }
1738
1739  }
1740
1741  }
1742
1743  }
1744
1745  }
1746
1747  }
1748
1749  }
1750
1751  }
1752
1753  }
1754
1755  }
1756
1757  }
1758
1759  }
1760
1761  }
1762
1763  }
1764
1765  }
1766
1767  }
1768
1769  }
1770
1771  }
1772
1773  }
1774
1775  }
1776
1777  }
1778
1779  }
1780
1781  }
1782
1783  }
1784
1785  }
1786
1787  }
1788
1789  }
1790
1791  }
1792
1793  }
1794
1795  }
1796
1797  }
1798
1799  }
1800
1801  }
1802
1803  }
1804
1805  }
1806
1807  }
1808
1809  }
1810
1811  }
1812
1813  }
1814
1815  }
1816
1817  }
1818
1819  }
1820
1821  }
1822
1823  }
1824
1825  }
1826
1827  }
1828
1829  }
1830
1831  }
1832
1833  }
1834
1835  }
1836
1837  }
1838
1839  }
1840
1841  }
1842
1843  }
1844
1845  }
1846
1847  }
1848
1849  }
1850
1851  }
1852
1853  }
1854
1855  }
1856
1857  }
1858
1859  }
1860
1861  }
1862
1863  }
1864
1865  }
1866
1867  }
1868
1869  }
1870
1871  }
1872
1873  }
1874
1875  }
1876
1877  }
1878
1879  }
1880
1881  }
1882
1883  }
1884
1885  }
1886
1887  }
1888
1889  }
1890
1891  }
1892
1893  }
1894
1895  }
1896
1897  }
1898
1899  }
1900
1901  }
1902
1903  }
1904
1905  }
1906
1907  }
1908
1909  }
1910
1911  }
1912
1913  }
1914
1915  }
1916
1917  }
1918
1919  }
1920
1921  }
1922
1923  }
1924
1925  }
1926
1927  }
1928
1929  }
1930
1931  }
1932
1933  }
1934
1935  }
1936
1937  }
1938
1939  }
1940
1941  }
1942
1943  }
1944
1945  }
1946
1947  }
1948
1949  }
1950
1951  }
1952
1953  }
1954
1955  }
1956
1957  }
1958
1959  }
1960
1961  }
1962
1963  }
1964
1965  }
1966
1967  }
1968
1969  }
1970
1971  }
1972
1973  }
1974
1975  }
1976
1977  }
1978
1979  }
1980
1981  }
1982
1983  }
1984
1985  }
1986
1987  }
1988
1989  }
1990
1991  }
1992
1993  }
1994
1995  }
1996
1997  }
1998
1999  }
2000
2001  }
2002
2003  }
2004
2005  }
2006
2007  }
2008
2009  }
2010
2011  }
2012
2013  }
2014
2015  }
2016
2017  }
2018
2019  }
2020
2021  }
2022
2023  }
2024
2025  }
2026
2027  }
2028
2029  }
2030
2031  }
2032
2033  }
2034
2035  }
2036
2037  }
2038
2039  }
2040
2041  }
2042
2043  }
2044
2045  }
2046
2047  }
2048
2049  }
2050
2051  }
2052
2053  }
2054
2055  }
2056
2057  }
2058
2059  }
2060
2061  }
2062
2063  }
2064
2065  }
2066
2067  }
2068
2069  }
2070
2071  }
2072
2073  }
2074
2075  }
2076
2077  }
2078
2079  }
2080
2081  }
2082
2083  }
2084
2085  }
2086
2087  }
2088
2089  }
2090
2091  }
2092
2093  }
2094
2095  }
2096
2097  }
2098
2099  }
2100
2101  }
2102
2103  }
2104
2105  }
2106
2107  }
2108
2109  }
2110
2111  }
2112
2113  }
2114
2115  }
2116
2117  }
2118
2119  }
2120
2121  }
2122
2123  }
2124
2125  }
2126
2127  }
2128
2129  }
2130
2131  }
2132
2133  }
2134
2135  }
2136
2137  }
2138
2139  }
2140
2141  }
2142
2143  }
2144
2145  }
2146
2147  }
2148
2149  }
2150
2151  }
2152
2153  }
2154
2155  }
2156
2157  }
2158
2159  }
2160
2161  }
2162
2163  }
2164
2165  }
2166
2167  }
2168
2169  }
2170
2171  }
2172
2173  }
2174
2175  }
2176
2177  }
2178
2179  }
2180
2181  }
2182
2183  }
2184
2185  }
2186
2187  }
2188
2189  }
2190
2191  }
2192
2193  }
2194
2195  }
2196
2197  }
2198
2199  }
2200
2201  }
2202
2203  }
2204
2
```



```
proc.c
25     {
26         initlock(&ptable.lock, "ptable");
27     }
28
29     // Must be called with interrupts disabled
30     int
31     cpuid() {
>32         return mycpu()-cpus;
33     }
34
35     // Must be called with interrupts disabled to avoid the caller being
36     // rescheduled between reading lapicid and running through the loop.
37     struct cpu*
38     mycpu(void)
39     {
40         int apicid, i;
41
remote Thread 1.1 In: cpuid                                     L32   PC: 0x801047b6
#1 0x803ff000 in ?? ()
#2 0x801047b6 in cpuid () at proc.c:32
#3 0x80117d48 in kpgdir ()
#4 0x800f5c80 in ?? ()
#5 0x80107d40 in seginit () at vm.c:24
#6 0x800f5d74 in ?? ()
#7 0x80103dfe in main () at main.c:24
(gdb) up 2
#2 0x801047b6 in cpuid () at proc.c:32
(gdb) |
```

بخش امتیازی (لینوکس)

پس از نصب اوبونتو 22.04 روی VirtualBox، دستور `uname -a` نشان می‌دهد که ورژن هسته لینوکس در این نسخه 5.15.0 است:

```
pasha@pasha-VirtualBox: ~
pasha@pasha-VirtualBox:~$ uname -a
Linux pasha-VirtualBox 5.15.0-50-generic #56-Ubuntu SMP Tue Sep 20 13:23:26 UTC
2022 x86_64 x86_64 x86_64 GNU/Linux
pasha@pasha-VirtualBox:~$
```

برای نزدیک بودن ورژن هسته جدید به ورژن قبلی، از کرنل 5.19.16 استفاده شد. همچنین برای کم حجم بودن هسته و صرف زمان کمتر هنگام کامپایل، از دستور `make defconfig` و پیکربندی پیش‌فرض آن استفاده کردیم. در نهایت پس از جایگزین کردن هسته، دستور `uname -a` ورژن جدید را نشان می‌دهد:

```
pasha@pasha-VirtualBox: ~
pasha@pasha-VirtualBox:~$ uname -a
Linux pasha-VirtualBox 5.19.16 #1 SMP PREEMPT_DYNAMIC Sun Oct 16 09:43:53 EDT 20
22 x86_64 x86_64 x86_64 GNU/Linux
pasha@pasha-VirtualBox:~$
```

برای نمایش اسم اعضای گروه در دستور `dmesg` یک فایل `group1.c` و یک `Makefile` به صورت زیر تهیه شده‌اند:

```
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("GPL");

int init_module(void)
{
    printk(KERN_INFO "Group 1:\n- Saman Eslami Nazari :
810199375\n- Pasha Barahimi      : 810199385\n- Misagh Mohaghegh
: 810199484\n");
    return 0;
}

void cleanup_module(void) {}

obj-m += group1.o

all:
    make -C /lib/modules/5.19.16/build M=$(PWD) modules
```

در نهایت پس از اجرای دستور `make`، یک فایل به نام `group1.ko` ایجاد می‌شود. پس از آن، از دستور `sudo insmod group1.ko` استفاده می‌کنیم. در نهایت اگر دستور `dmesg` را اجرا کنیم، می‌توانیم اسم اعضای گروه را در انتهای خروجی این دستور مشاهده کنیم:

```
[ 1092.437421] group1: loading out-of-tree module taints kernel.  
[ 1092.439510] Group 1:  
- Saman Eslami Nazari : 810199375  
- Pasha Barahimi : 810199385  
- Misagh Mohaghegh : 810199484
```