Repository Link: <https://github.com/SamanEN/Operating-System-Lab-Projects>

Latest Commit Hash: <hash>

# آشنایی با سیستم‌عامل xv6

## معماری سیستم‌عامل 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 داکیومنت 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 file systems، system calls، user-level و... می‌شود نیز قابل مشاهده است.

## بخش های پردازه و چگونگی اختصاص پردازنده به پردازه های مختلف

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

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

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

## مفهوم file descriptor و عملکرد pipe در xv6

## توابع exec و fork

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

* pid = 0: در پردازه فرزند هستیم.
* pid > 0: در پردازه پدر هستیم و مقدار pid در واقع آی‌دی پردازه فرزند است.
* pid < 0: در زمان اجرای تابع fork و پردازه جدید اروری وجود داشته و پردازه فرزند ایجاد نشده است.

اگر پس از fork کردن از تابع wait(int\*) استفاده شود، پردازه پدر منتظر پایان یافتن پردازه فرزند می‌شود و سپس کار خود را ادامه می‌دهد. خروجی این تابع، pid پردازه پایان یافته است. اگر پردازه فعلی هیچ پردازه فرزندی نداشته باشد، خروجی این تابع -1 خواهد بود. همچنین ورودی این تابع پوینتر به متغیری است که در نهایت status code مروبط به پردازه 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 پردازه 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 پردازه 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 پردازه 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 پردازه جدید ایجاد می‌کند.
3. در پردازه فرزند با استفاده از تابع exec برنامه درخواست شده توسط کاربر را جایگزین پردازه فعلی (فرزند) می‌کند.
4. در پردازه پدر برای اتمام کار پردازه فرزندwait می‌کند.
5. پس از اتمام پردازه فرزند به main باز می‌گردد و منتظر دستور جدید می‌شود.

زمانی که کاربر برای یک دستور از redirection استفاده می‌کند، تغییرات لازم در file descriptorها پس از fork و پیش از exec و در پردازه 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 دردسرهای خودش را دارد و یا اینکه shell پیش از اجرای این تابع،  
file descriptorهای 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



پس از بوت شدن سیستم‌عامل 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

## 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 می‌شود:

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);  
}

## 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’)، می‌توان 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);  
}

## 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;

متغییر 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 متغییر بعدی برای نگه داشتن دستور اصلی است که کاربر قبل از دریافت پیشنهادات وارد کرده بود؛ استفاده از این متغییر ها 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 ارئه 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



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 سیستم‌عامل و xv6

## سه وظیفه اصلی سیستم‌عامل

## گروه های فایل های اصلی xv6

# کامپایل سیستم‌عامل xv6

## دستور make -n و کدام دستور فایل نهایی را می‌سازد؟

## متغیر های 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 هدف[[1]](#footnote-1) با پیشنیاز[[2]](#footnote-2)های فایل آبجکت هدف (file\_name.o) و متغیر ULIB دارد. بنابراین هدف‌های موجود در UPROGS منجر به ساخت فایل آبجکت برنامه‌های کاربر، اجرا شدن هدف‌های مربوط به ULIB می‌شود و در نهایت اجرای دستور ld می‌شود. دستور ld برای پیوند[[3]](#footnote-3) فایل‌های مورد نیاز و تولید یک فایل قابل اجرا مورد استفاده قرار می‌گیرد. علاوه بر آن فایل‌های آبجکت مربوط به هر برنامه (file\_name.o) توسط یک قانون درونی[[4]](#footnote-4) 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، strcmp، strcpy، 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

## محتوای دو دیسک ورودی QEMU

# مراحل بوت سیستم‌عامل xv6

# اجرای بوت‌لودر

## 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.

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

همه فایل‌های باینری آبجکت xv6 در فرمت ELF[[5]](#footnote-5) 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

تولید می‌شود. این فلگ‌های objcopy در [بخش بعدی](#_علت_استفاده_از) 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.

برای تبدیل bootblock به اسمبلی، از کامند زیر استفاده می‌کنیم:

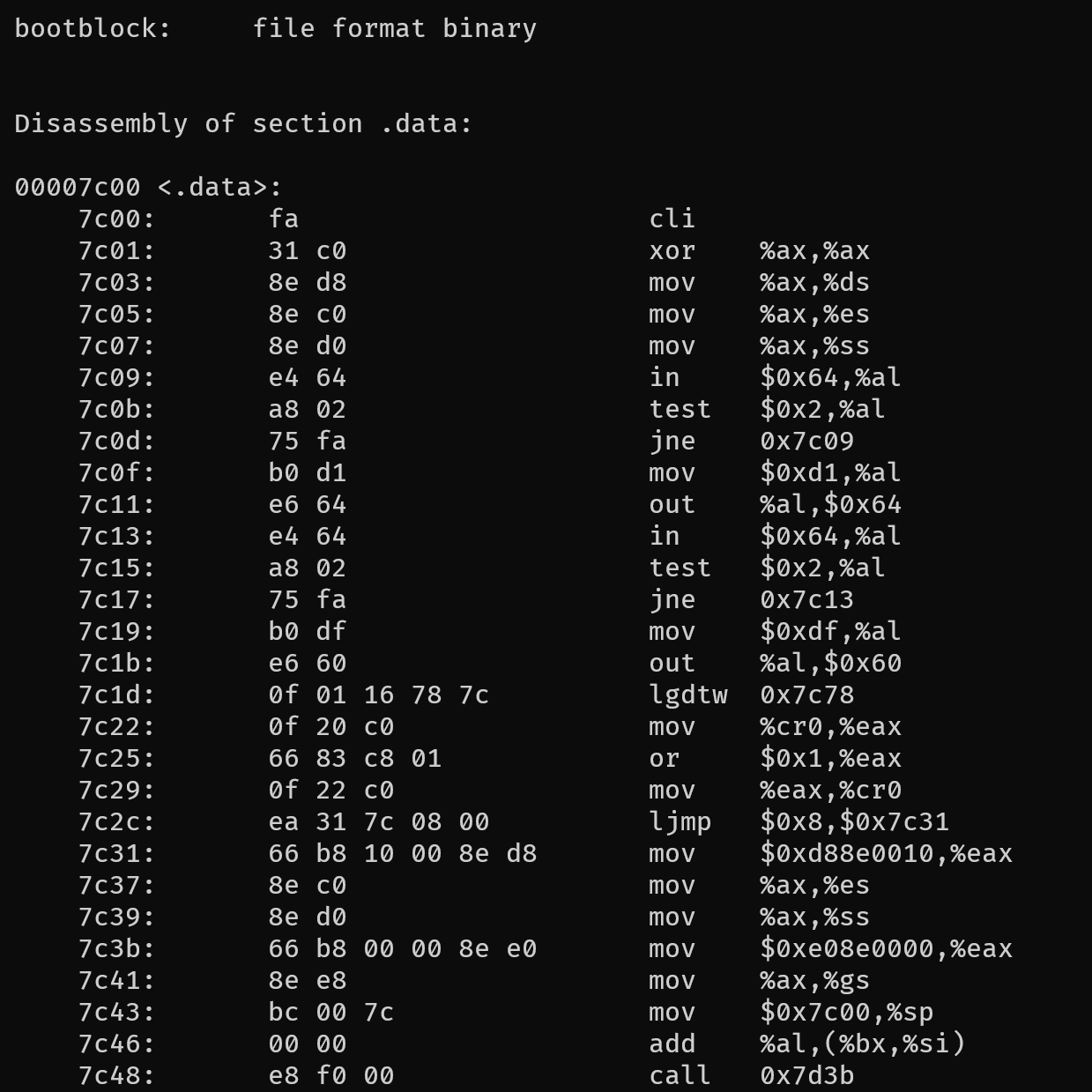
objdump -D -b binary -m i386 -M addr16,data16 bootblock

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

* -D : برای disassemble کردن باینری.
* -b binary : نوع فایل را raw binary در نظر می‌گیریم.
* -m i386 : معماری اسمبلی فایل را مشخص می‌کنیم.
* -M addr16,data16 : از آنجا که وقتی BIOS سکتور بوت را لود می‌کند در real mode هستیم و CPU در حالت 16بیت است، اسمبلی 16بیت نیز استفاده شده است پس هنگام disassemble کردن هم می‌گوییم که آدرس‌ها و داده‌ها را 16بیت در نظر بگیرد.

می‌توانیم 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 می‌بینیم که ابتدای آن بسیار مشابه با bootasm.S است:



## 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 مروبط 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.
* -O : این آپشن نوع فرمت فایل مقصد را نشان می‌دهد. برای مثال با استفاده از آپشن -O binary 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.

## 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 16بیت است و فقط 1 مگابایت حافظه داریم.

برای اینکه بتوانیم از پردازنده 32بیت 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.

## وظیفه ثبات های x86

* ثبات عام منظوره: پردازنده‌های x86 دارای 8 ثبات عام منظوره هستند. از این ثبات‌ها می‌توان به ثبات انباشت‌کننده[[6]](#footnote-6) 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 پشته[[7]](#footnote-7) می‌باشد. این ثبات اطلاعاتی مربوط به قطعه‌ای از حافظه را ذخیره می‌کند که از آن برای پشته فراخوانی[[8]](#footnote-8) 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](https://stackoverflow.com/questions/8642813/what-is-the-difference-between-esp-register-and-ss-register) 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 32بیتی و 64بیتی می‌باشند. هر بیت از این ثبات نشان‌دهنده یک پرچم[[9]](#footnote-9) 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 |
| 10 | DF | Direction flag | کنترل |
| 11 | OF | Overflow flag | وضعیت |
| 12-13 | IOPL | سطح دسترسی ورودی خروجی | سیستم |
| 14 | NT | پرچم فعالیت تو در تو | سیستم |
| 15 |  | رزرو شده |  |

* 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:

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

## نقص اصلی real mode پردازنده x86

## آدرس دهی حافظه در real mode

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

## 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](https://github.com/torvalds/linux/blob/master/arch/x86/entry/entry.%20S)

که برای [32بیت](https://github.com/torvalds/linux/blob/master/arch/x86/entry/entry_32.S) و [64بیت](https://github.com/torvalds/linux/blob/master/arch/x86/entry/entry_64.S) جدا است.

# اجرای هسته xv6

## چرا آدرس فیزیکی است؟

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.

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

## مختصری راجع به محتوای فضای آدرس مجازی هسته

## چرا برای کد و داده های سطح کاربر پرچم SEG\_USER تنظیم شده است؟

قطعه‌بندی در xv6 در تابع seginit 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[[10]](#footnote-10) مشخص شده که این دسکتریپتور 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 کدِ دسکریپتورِ 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[[11]](#footnote-11) نیز از روی سطح دسترسی دسکریپتور یا همان DPL[[12]](#footnote-12) مشخص می‌شود. بدین گونه از طریق DPL متفاوت می‌توان سطح دسترسی فعلی دستورالعمل‌ها را نیز تعیین کرد؛ حتی اگر این دسکتریپتورها 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 فعلی (CPL) برابر مقدار سطح دسترسی دسکتریپتور (DPL) قطعه‌ای است که کد مربوط به این دستور العمل در آن قرار گرفته است و اگر این دستورالعمل 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 دستورالعملی را نمی‌تواند اجرا کند.

# اجرای نخستین برنامه سطح کاربر

## اجزای struct proc و معادل آن در لینوکس

## چرا به خواب رفتن در کد مدیریت‌کننده سیستم مشکل ساز است؟

## تفاوت فضای آدرس هسته با فضای آدرس توسط kvmalloc

## تفاوت فضای آدرس inituvm با فضای آدرس کاربر در کد مدیریت سیستم

## کدام بخش از آماده‌سازی سیستم بین تمامی هسته های پردازنده مشترک و کدام بخش اختصاصی است؟

هسته اول که فرآیند بوت را انجام می‌دهد توسط کد entry.S وارد تابع main در فایل main.c می‌شود. تمامی توابع آماده سازی سیستم که در این تابع فراخوانده شده‌اند توسط این هسته اجرا می‌شوند. از طرفی، هسته‌های دیگر از طریق کد entryother.S وارد تابع mpenter می‌شوند. در این تابع نیز 4 تابع برای آماده‌سازی فراخوانده می‌شوند. در نتیجه می‌توان گفت این 4 تابع بین تمامی هسته‌ها مشترک خواهند بود. یکی از این توابع به نام switchkvm به صورت مستقیم با هسته اول مشترک نیست. این تابع در mpenter صدا زده می‌شود در صورتی که در تابع main وجود ندارد. در واقع تابع kvmalloc که در main صدا زده می‌شود به صورت زیر است:

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

خط اول تابع یک page table برای کرنل ایجاد می‌کند که این مورد توسط هسته اول انجام می‌پذیرد. پس از آن باید هسته به این page table سوییچ کند که این کار در تمامی هسته‌ها انجام می‌پذیرد. بخش‌هایی از آماده‌سازی سیستم که در تمام هسته‌ها مشترک هستند به شرح زیر است:

* switchkvm
* seginit
* lapicinit
* mpmain

همچنین بخش‌هایی که تنها در هسته اول (به صورت اختصاصی) اجرا می‌شوند به شرح زیر است:

* kinit1
* kvmalloc (setupkvm)
* mpinit
* picinit
* ioapicinit
* consoleinit
* uartinit
* pinit
* tvinit
* binit
* fileinit
* ideinit
* startothers
* kinit2
* userinit

از موارد اختصاصی هسته اول می‌توان به تابع startothers اشاره کرد که واضح است فقط پردازنده اول نیاز است بقیه پردازنده‌ها 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 داکیومنت 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.

## برنامه معادل initcode.s در هسته لینوکس

# اشکال زدایی

# روند اجرای GDB

## دستور مشاهده breakpoint ها

## دستور حذف یک breakpoint

# کنترل روند اجرا و دسترسی به حالت سیستم

## خروجی bt

## تفاوت دستور x و print

## نمایش وضعیت ثبات ها و متغیر های محلی؛ رجیستر های edi و esi

## ساختار struct input

# اشکال زدایی در سطح کد اسمبلی

## خروجی دستور های layout src و layout asm در TUI

## دستور های جابجایی میان توابع زنجیره فراخوانی جاری (نقطه توقف)

1. Target [↑](#footnote-ref-1)
2. Prerequisite [↑](#footnote-ref-2)
3. Link [↑](#footnote-ref-3)
4. Built-in implicit rule [↑](#footnote-ref-4)
5. Executable and Linkable Format [↑](#footnote-ref-5)
6. Accumulator register (AX) [↑](#footnote-ref-6)
7. Stack [↑](#footnote-ref-7)
8. Call stack [↑](#footnote-ref-8)
9. Flag [↑](#footnote-ref-9)
10. Global descriptor table [↑](#footnote-ref-10)
11. Current privilege level [↑](#footnote-ref-11)
12. Descriptor privilege level [↑](#footnote-ref-12)