



به نام خدا

# پروژه اول آزمایشگاه سیستم عامل

(آشنایی با هسته سیستم عامل xv6)



## مقدمه

سیستم عامل xv6 یک سیستم عامل آموزشی است که در سال 2006 توسط محققان دانشگاه MIT به وجود آمده است. این سیستم عامل به زبان C و با استفاده از هسته Unix Version 6 نوشته شده و بر روی معماری Intel x86 قابل اجرا می باشد. سیستم عامل xv6 علی رغم سادگی و حجم کم، نکات اساسی و مهم در طراحی سیستم عامل را دارا است و برای مقاصد آموزشی بسیار مفید می باشد. تا پیش از این، در درس سیستم عامل دانشگاه تهران از هسته سیستم عامل لینوکس استفاده می شد که پیچیدگی های زیادی دارد. در ترم پیش رو، دانشجویان آزمایشگاه سیستم عامل بایستی پروژه های مربوطه را بر روی سیستم عامل xv6 اجرا و پیاده سازی نمایند. در این پروژه، ضمن آشنایی به معماری و برخی نکات پیاده سازی سیستم عامل، آن را اجرا و اشکال زدایی خواهیم کرد و همچنین برنامه ای در سطح کاربر خواهیم نوشت که بر روی این سیستم عامل قابل اجرا باشد.

## آشنایی با سیستم عامل xv6

کدهای مربوط به سیستم عامل xv6 از لینک زیر قابل دسترسی است:

<https://github.com/mit-pdos/xv6-public>

همچنین مستندات این سیستم عامل و فایل شامل کدهای آن نیز در صفحه درس بارگذاری شده است. برای این پروژه، نیاز است که فصل های 0 و 1 از مستندات فوق را مطالعه کرده و به برخی سوالات منتخب پاسخ دهید. پاسخ این سوالات را در قالب یک گزارش بارگذاری خواهید کرد.

1. معماری سیستم عامل xv6 چیست؟ چه دلایلی در دفاع از نظر خود دارید؟
2. یک پردازنده<sup>1</sup> در سیستم عامل xv6 از چه بخش هایی تشکیل شده است؟ این سیستم عامل به طور کلی چگونه پردازنده را به پردازنده های مختلف اختصاص میدهد؟
3. مفهوم file descriptor در سیستم عامل های مبتنی بر UNIX چیست؟ عملکرد pipe در سیستم عامل xv6 چگونه است و به طور معمول برای چه هدفی استفاده می شود؟
4. فراخوانی های سیستمی exec و fork چه عملی انجام میدهند؟ از نظر طراحی، ادغام نکردن این دو چه مزیتی دارد؟

## اجرا و اشکال زدایی

در این بخش به اجرای سیستم عامل xv6 خواهیم پرداخت. علی رغم اینکه این سیستم عامل قابل اجرای مستقیم بر روی سخت افزار است، به دلیل آسیب پذیری بالا و رعایت مسائل ایمنی از این کار اجتناب نموده و سیستم عامل را به کمک برابرساز<sup>2</sup> Qemu روی سیستم عامل لینوکس اجرا میکنیم. برای این منظور لازم است که کدهای مربوط به سیستم عامل را از لینک ارائه شده clone و یا دانلود کنیم. در ادامه با اجرای دستور make در پوشه دانلود، سیستم عامل کامپایل می شود. در نهایت با اجرای دستور make qemu سیستم عامل بر روی برابرساز اجرا میشود (توجه شود که فرض شده Qemu از قبل بر روی سیستم عامل شما نصب بوده است. در غیر این صورت ابتدا آن را نصب نمایید).

<sup>1</sup> Process  
<sup>2</sup> Emulator

## اضافه کردن یک متن به Boot Message

در این بخش، شما باید نام اعضای گروه را پس از بوت شدن سیستم عامل روی ماشین مجازی Qemu، در انتهای پیام‌های نمایش داده شده در کنسول نشان دهید. تصویر این اطلاعات را در گزارش خود قرار دهید.

### اضافه کردن چند قابلیت به کنسول xv6

در این قسمت می‌خواهیم چند قابلیت کاربردی به کنسول xv6 اضافه کنیم. پس از اجرای سیستم عامل بر روی Qemu، در صورت استفاده از کلیدهای Ctrl+B، Ctrl+F، Ctrl+L و ↑ و ↓، معادل کاراکتری آن‌ها، در کنسول چاپ می‌شود. کد xv6 را به نحوی تغییر دهید تا قابلیت‌های زیر در آن پیاده‌سازی شده باشد:

1. در صورتی که کاربر دستور Ctrl+B را وارد کرد، نشانگر در ترمینال به اندازه یک واحد (یک کاراکتر) به عقب (سمت چپ) حرکت می‌کند و پس از آن، تمام تغییرات وارده توسط کاربر در محل جدید باید اثر داده بشوند.
2. در صورتی که کاربر دستور Ctrl+F را وارد کرد، نشانگر در ترمینال به اندازه یک واحد (یک کاراکتر) به جلو (سمت راست) حرکت می‌کند و پس از آن، تمام تغییرات وارده توسط کاربر در محل جدید باید اثر داده بشوند.
3. در صورتی که کاربر دستور Ctrl+L را وارد کرد، صفحه فعلی در ترمینال پاک شود و بجز علامت \$ بیانگر شروع فرمان، قرار گرفته در اولین خط ترمینال، چیز دیگری نمایش داده نشود.
4. اگر کاربر کلید ↑ را روی کیبورد فشرد، آخرین دستور وارد شده باید نمایش داده شود. اگر دو بار کلید ↑ را فشرد، یکی مانده به آخرین دستور وارد شده باید نمایش داده شود. اگر سه بار کلید ↑ را وارد کرد، دو تا مانده به آخرین دستور و به همین صورت الی آخر (۱۰ دستور آخر بایستی نگهداری شوند). توجه شود نیامندی عملکرد این دستور، حداقل وجود یک دستور قبل تر می‌باشد و لذا در شروع سیستم عامل و یا رسیدن به آخرین دستور لیست، در صورت استفاده، بر روی دستور فعلی باقی می‌ماند.
5. اگر کاربر کلید ↓ را روی کیبورد فشرد، دستور بعدی وارد شده باید نمایش داده شود و با تکرار این عمل، هربار دستور بعدتر نمایش داده شود. توجه شود که لازمه عملکرد این قابلیت، وجود دستور بعدی می‌باشد، یعنی باید حداقل یکبار از ↑ قبل از آن استفاده شده باشد.

- توجه شود که علاوه بر نمایش درست بر روی کنسول، باید دستورات نوشته شده با کلیدهای ترکیبی فوق، قابلیت اجرای درست را نیز داشته باشند.

## اجرا و پیاده‌سازی یک برنامه سطح کاربر

در این قسمت شما باید یک برنامه سطح کاربر و به زبان C بنویسید و به برنامه‌های سطح کاربر سیستم عامل اضافه کنید. نام این برنامه strdiff می‌باشد. این برنامه دو کلمه هر کدام با طول کمینه 1 تا سقف 15 کاراکتر از حروف انگلیسی غیرحساس به بزرگی یا کوچکی، از ورودی دریافت می‌کند و اختلاف رشته‌ای رشته اول با رشته دوم را بصورت یک رشته باینری محاسبه می‌کند. اختلاف رشته‌ای به این صورت محاسبه می‌شود که از ابتدای هر دو رشته شروع می‌کنیم و حرف به حرف با هم مقایسه می‌کنیم و در یک رشته باینری نتیجه مقایسه‌ها را نگه می‌داریم؛ به این صورت که اگر حرف i ام رشته اول از حرف i ام رشته دوم بزرگتر یا مساوی باشد،

رقم 0 و در غیر این صورت رقم 1 در جایگاه i ام حاصل باینری قرار می‌گیرد. همچنین اگر طول یکی از رشته‌ها از رشته دیگر بلندتر بود، حروف مازاد آن رشته در مقایسه‌هایشان به عنوان حرف بزرگتر به حساب می‌آیند. بزرگتر بودن حروف نسبت به یکدیگر هم به این صورت تعریف می‌شود که یک حرف در حروف الفبای انگلیسی بعد از حرف دیگر آمده باشد. به عنوان مثال حرف f از حرف b بزرگتر است. در نهایت، خروجی محاسبه را در یک فایل تکست با نام `strdiff_result.txt` ذخیره کنید. اگر فایل تکست از قبل موجود بود، جواب بر روی آن بازنویسی شود.

```
$ strdiff apple banana
```

```
$ cat strdiff_result.txt
```

```
100011
```

از دستورات `open`، `read`، `write` و `close` استفاده کنید که برای باز کردن، خواندن، نوشتن و بستن فایل‌ها استفاده می‌شود. برای پیاده‌سازی این برنامه سطح کاربر، علاوه بر نوشتن کد، باید در فایل `Makefile` نیز تغییرات لازم را بوجود آورید تا این برنامه مثل دستورات دیگر از قبیل `ls` اجرا شود.

## مقدمه‌ای درباره سیستم عامل و xv6

سیستم عامل جزو نخستین نرم‌افزارهایی است که پس از روشن شدن سیستم، اجرا می‌گردد. این نرم‌افزار، رابط نرم‌افزارهای کاربردی با سخت‌افزار رایانه است.

5. سه وظیفه اصلی سیستم عامل را نام ببرید.

6. فایل‌های اصلی سیستم عامل `xv6` در صفحه یک کتاب `xv6` لیست شده‌اند. به طور مختصر هر گروه را

توضیح دهید. نام پوشه اصلی فایل‌های هسته سیستم عامل، فایل‌های سرانند<sup>3</sup> و فایل سیستم در

سیستم عامل لینوکس چیست؟ در مورد محتویات آن مختصراً توضیح دهید.

## کامپایل سیستم عامل xv6

یکی از روش‌های متداول کامپایل و ایجاد نرم‌افزارهای بزرگ در سیستم عامل‌های مبتنی بر `Unix` استفاده از ابزار `Make` است. این ابزار با پردازش فایل‌های موجود در کد منبع برنامه، موسوم به `Makefile`، شیوه کامپایل و لینک فایل‌های دودویی به یکدیگر و در نهایت ساختن کد دودویی نهایی برنامه را تشخیص می‌دهد. ساختار `Makefile` قواعد خاص خود را داشته و می‌تواند بسیار پیچیده باشد. اما به طور کلی شامل قواعد<sup>4</sup> و متغیرها<sup>5</sup> می‌باشد. در `xv6` تنها یک `Makefile` وجود داشته و تمامی فایل‌های سیستم عامل نیز در یک پوشه قرار دارند. بیلد سیستم عامل از طریق دستور `make-j8` در پوشه سیستم عامل صورت می‌گیرد.

7. دستور `make -n` را اجرا نمایید. کدام دستور، فایل نهایی هسته را می‌سازد؟

8. در `Makefile` متغیرهایی به نام‌های `UPROGS` و `ULIB` تعریف شده است. کاربرد آن‌ها چیست؟

<sup>3</sup> Header Files

<sup>4</sup> Rules

<sup>5</sup> Variables

## اجرا بر روی شبیه ساز QEMU

xv6 قابل اجرا بر روی سخت افزار واقعی نیز است. اما اجرا بر روی شبیه ساز قابلیت ردگیری و اشکال زدایی بیشتری ارائه می کند. جهت اجرای سیستم عامل بر روی شبیه ساز، کافی است دستور `make qemu` در پوشه سیستم عامل اجرا گردد.

9. دستور `make qemu -n` را اجرا نمایید. دو دیسک به عنوان ورودی به شبیه ساز داده شده است. محتوای آن ها چیست؟ (راهنمایی: این دیسک ها حاوی سه خروجی اصلی فرایند بیلد هستند).

## مراحل بوت سیستم عامل xv6

### اجرای بوت لودر

هدف از بوت آماده سازی سیستم عامل برای سرویس دهی به برنامه های کاربر است. پس از بوت، سیستم عامل سازوکاری جهت ارائه سرویس به برنامه های کاربردی خواهد داشت که این برنامه ها بدون هیچ مزاحمتی بتوانند از آن استفاده نمایند. کوچکترین واحد دسترسی دیسک ها در رایانه های شخصی سکتور<sup>6</sup> است. در این جا هر سکتور ۵۱۲ بایت است. اگر دیسک قابل بوت باشد، نخستین سکتور آن سکتور بوت<sup>7</sup> نام داشته و شامل بوت لودر<sup>8</sup> خواهد بود. بوت لودر کدی است که سیستم عامل را در حافظه بارگذاری می کند. یکی از روش های راه اندازی اولیه رایانه، بوت مبتنی بر سیستم ورودی/خروجی مقدماتی<sup>9</sup> (BIOS) است. BIOS در صورت یافتن دیسک قابل بوت، سکتور نخست آن را در آدرس 0x7C00 از حافظه فیزیکی کپی نموده و شروع به اجرای آن می کند.

10. در xv6 در سکتور نخست دیسک قابل بوت، محتوای چه فایلی قرار دارد. (راهنمایی: خروجی دستور `make -n` را بررسی نمایید).

11. برنامه های کامپایل شده در قالب فایل های دودویی نگهداری می شوند. فایل مربوط به بوت نیز دودویی است. نوع این فایل دودویی چیست؟ تفاوت این نوع فایل دودویی با دیگر فایل های دودویی کد xv6 چیست؟ چرا از این نوع فایل دودویی استفاده شده است؟ این فایل را به زبان قابل فهم انسان (اسمبلی) تبدیل نمایید. (راهنمایی: از ابزار `objdump` استفاده کنید. باید بخشی از آن مشابه فایل `bootasm.S` باشد).

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

13. بوت سیستم توسط فایل های `bootasm.S` و `bootmain.c` صورت می گیرد. چرا تنها از کد C استفاده نشده است؟

معماری سیستم شبیه سازی شده x86 است. حالت سیستم در حال اجرا در هر لحظه را به طور ساده می توان شامل حالت پردازنده و حافظه دانست. بخشی از حالت پردازنده در ثبات های آن نگهداری می شود.

14. یک ثبات عام منظوره<sup>10</sup>، یک ثبات قطعه<sup>11</sup>، یک ثبات وضعیت<sup>12</sup> و یک ثبات کنترلی<sup>13</sup> در معماری x86 را نام برده و وظیفه هر یک را به طور مختصر توضیح دهید.

<sup>6</sup> Sector

<sup>7</sup> Boot Sector

<sup>8</sup> Boot Loader

<sup>9</sup> Basic Input/Output System

<sup>10</sup> General Purpose Register

<sup>11</sup> Segment Register

<sup>12</sup> Status Registers

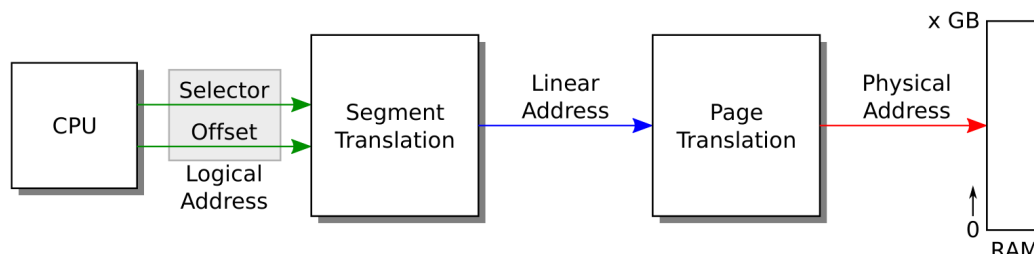
<sup>13</sup> Control Registers

وضعیت ثبات‌ها را می‌توان به کمک gdb و دستور info registers مشاهده نمود. وضعیت برخی از ثبات‌های دیگر نیاز به دسترسی ممتاز<sup>14</sup> دارد. به این منظور می‌توان از qemu استفاده نمود. کافی است با زدن Ctrl + A و سپس C به ترمینال qemu رفته و دستور info registers را وارد نمود. با تکرار همان دکمه‌ها می‌توان به xv6 بازگشت.

15. پردازنده‌های x86 دارای مدهای مختلفی هستند. هنگام بوت، این پردازنده‌ها در مد حقیقی<sup>15</sup> قرار داده می‌شوند. مدی که سیستم عامل ام‌اس‌داس<sup>16</sup> (MS DOS) در آن اجرا می‌شد. چرا؟ یک نقص اصلی این مد را بیان نمایید؟

16. آدرس‌دهی به حافظه در این مد شامل دو بخش قطعه<sup>17</sup> و افسست<sup>18</sup> بوده که اولی ضمنی و دومی به طور صریح تعیین می‌گردد. به طور مختصر توضیح دهید.

در ابتدا qemu یک هسته را جهت اجرای کد بوت bootasm.S فعال می‌کند. فرایند بوت در بالاترین سطح دسترسی<sup>19</sup> صورت می‌گیرد. به عبارت دیگر، بوت‌لودر امکان دسترسی به تمامی قابلیت‌های سیستم را دارد. در ادامه هسته به مد حفاظت‌شده<sup>20</sup> تغییر مد می‌دهد (خط ۹۱۵۳). در مد حفاظت‌شده، آدرس مورد دسترسی در برنامه (آدرس منطقی) از طریق جداولی به آدرس فیزیکی حافظه<sup>21</sup> نگاشت پیدا می‌کند. ساختار آدرس‌دهی در این مد در شکل زیر نشان داده شده است.



هر آدرس در کد برنامه یک آدرس منطقی<sup>22</sup> است. این آدرس توسط سخت افزار مدیریت حافظه در نهایت به یک آدرس فیزیکی در حافظه نگاشت داده می‌شود. این نگاشت دو بخش دارد: ۱) ترجمه قطعه<sup>23</sup> و ۲) ترجمه صفحه<sup>24</sup>. مفهوم ثبات‌های قطعه در این مد تا حد زیادی با نقش آن‌ها در مد حقیقی متفاوت است. این ثبات‌ها با تعامل با جدولی تحت عنوان جدول توصیف‌گر سراسری<sup>25</sup> (GDT) ترجمه قطعه را انجام می‌دهند. به این ترتیب ترجمه آدرس در مد محافظت‌شده بسیار متفاوت خواهد بود. در بسیاری از سیستم‌عامل‌ها از جمله xv6 و لینوکس ترجمه قطعه یک نگاشت همانی است. یعنی GDT به نحوی مقداردهی می‌گردد (خطوط ۹۱۸۲ تا ۹۱۸۵) که می‌توان از گزینش‌گر<sup>26</sup> صرف‌نظر نموده و افسست را به عنوان آدرس منطقی در نظر گرفت و این افسست را دقیقاً به عنوان آدرس خطی<sup>27</sup> نیز در نظر گرفت. به عبارت دیگر می‌توان فرض نمود که آدرس‌ها دوبخشی نبوده و صرفاً یک عدد هستند. یک آدرس برنامه (مثلاً آدرس یک اشاره‌گر یا آدرس قطعه‌ای از کد برنامه) یک آدرس منطقی (و همین‌طور در این‌جا یک آدرس خطی) است. به عنوان مثال در خط ۹۲۲۴ آدرس

<sup>14</sup> Privileged Access

<sup>15</sup> Real Mode

<sup>16</sup> Microsoft Disk Operating System

<sup>17</sup> Segment

<sup>18</sup> Offset

<sup>19</sup> سطوح دسترسی در ادامه پروژه توضیح داده خواهد شد.

<sup>20</sup> Protected Mode

<sup>21</sup> منظور از آدرس فیزیکی یک آدرس بکتا در سخت‌افزار حافظه است که پردازنده به آن دسترسی پیدا می‌کند.

<sup>22</sup> Logical Address

<sup>23</sup> Segment Translation

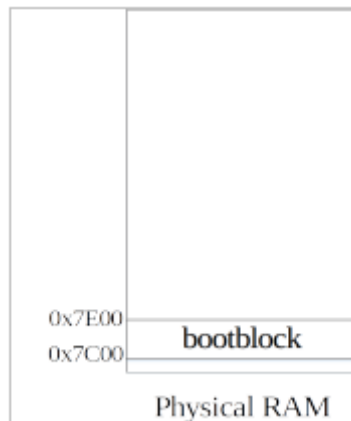
<sup>24</sup> Page Translation

<sup>25</sup> Global Descriptor Table

<sup>26</sup> Selector

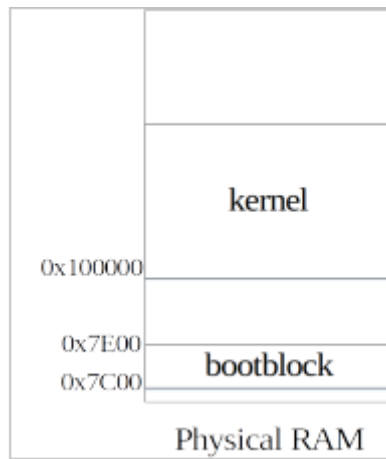
<sup>27</sup> Linear Address

اشاره گر elf که به 0x10000 مقداردهی شده است یک آدرس منطقی است. به همین ترتیب آدرس تابع bootmain() که در زمان کامپایل تعیین می گردد نیز یک آدرس منطقی است. در ادامه بنابر دلایل تاریخی به آدرس هایی که در برنامه استفاده می شوند، آدرس مجازی<sup>28</sup> اطلاق خواهد شد. نگاشت دوم یا ترجمه صفحه در کد بوت فعال نمی شود. لذا در این جا نیز نگاشت همانی وجود داشته و به این ترتیب آدرس مجازی برابر آدرس فیزیکی خواهد بود. نگاشت آدرس ها (و عدم استفاده مستقیم از آدرس فیزیکی) اهداف مهمی را دنبال می کند که در فصل مدیریت حافظه مطرح خواهد شد. از مهمترین این اهداف، حفاظت محتوای حافظه برنامه های کاربردی مختلف از یکدیگر است. بدین ترتیب در لحظه تغییر مد، وضعیت حافظه (فیزیکی) سیستم به صورت شکل زیر است.



17. کد bootmain.c هسته را با شروع از سکتور بعد از سکتور بوت خوانده و در آدرس 0x100000 قرار می دهد.<sup>29</sup> علت انتخاب این آدرس چیست؟

حالت حافظه پس از این فرایند به صورت شکل زیر است.



به این ترتیب در انتهای بوت، کد هسته سیستم عامل به طور کامل در حافظه قرار گرفته است. در گام انتهایی، بوت لودر اجرا را به هسته واگذار می نماید. باید کد ورود به هسته اجرا گردد. این کد اسمبلی در فایل entry.S قرار داشته و نماد (بیانگر مکانی از کد) entry از آن فراخوانی می گردد. آدرس این نماد در هسته بوده و حدود 0x100000 است.

18. کد معادل entry.S در هسته لینوکس را بیابید.

<sup>28</sup> Virtual Address

<sup>29</sup> دقت شود آدرس 0x100000 تنها برای خواندن هدر فایل elf استفاده شده است و محتوای فایل هسته در 0x100000 که توسط paddr (مخفف آدرس فیزیکی) تعیین شده است، کپی می شود. این آدرس در زمان لینک توسط kernel.ld تعیین شده و در فایل دودویی در قالب خاصی قرار داده شده است.

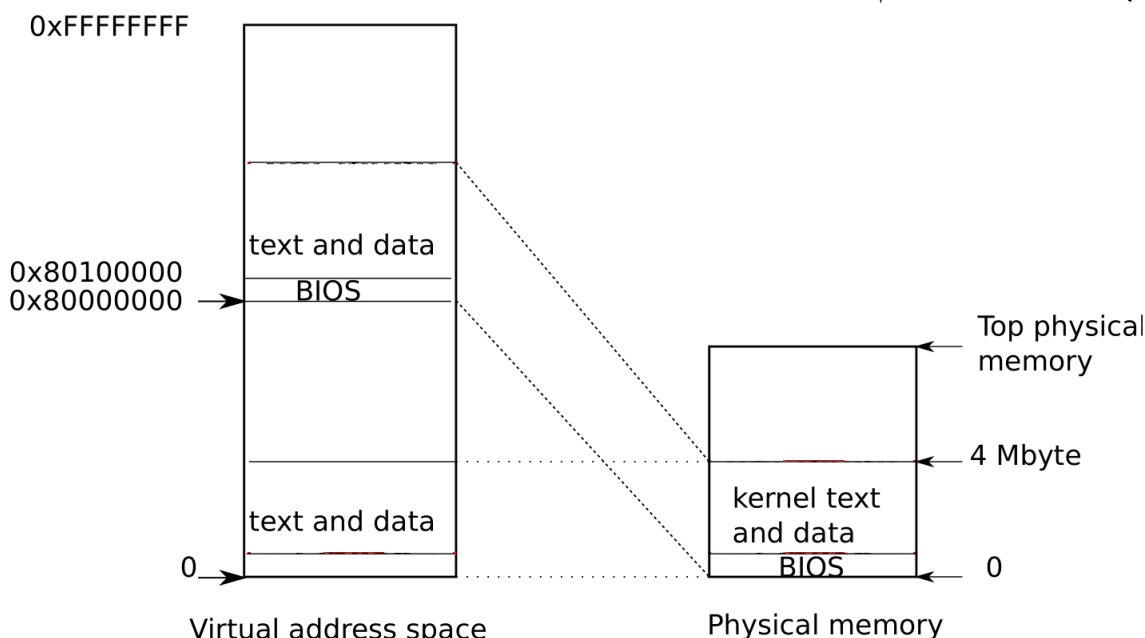


## اجرای هسته xv6

هدف از `entry.S` ورود به هسته و آماده‌سازی جهت اجرای کد `C` آن است. در شرایط کنونی نمی‌توان کد هسته را اجرا نمود. زیرا به گونه‌ای لینک شده است که آدرس‌های مجازی آن بزرگتر از `0x80100000` هستند. می‌توان این مسئله را با اجرای دستور `cat kernel.sym` بررسی نمود. در همین راستا نگاشت مربوط به صفحه‌بندی<sup>30</sup> (ترجمه صفحه) از حالت همانی خارج خواهد شد. در صفحه‌بندی، هر کد در حال اجرا بر روی پردازنده، از جدولی برای نگاشت آدرس مورد استفاده‌اش به آدرس فیزیکی استفاده می‌کند. این جدول خود در حافظه فیزیکی قرار داشته و یک آدرس فیزیکی مختص خود را دارد. در حین اجرا این آدرس در ثبات کنترلی `cr3` بارگذاری شده<sup>31</sup> و به این ترتیب پردازنده از محل جدول نگاشت‌های جاری اطلاع خواهد داشت.

19. چرا این آدرس فیزیکی است؟

جزئیات جدول نگاشت‌ها پیچیده است. به طور ساده این جدول دارای مدخل‌هایی است که تکه‌ای پیوسته از حافظه مجازی (یا خطی با توجه به خنثی شدن تأثیر آدرس منطقی) را به تکه‌ای پیوسته به همین اندازه از حافظه فیزیکی نگاشت می‌دهد. این اندازه‌ها در هر معماری، محدود هستند. به عنوان مثال در `entry.S` دو تکه پیوسته چهار مگابایتی از حافظه خطی به دو تکه پیوسته چهار مگابایتی از حافظه فیزیکی نگاشت داده شده است. هر تکه پیوسته یک صفحه<sup>32</sup> نام دارد. یعنی حالت حافظه مطابق شکل زیر خواهد بود.



نیمه چپ شکل، فضای آدرس مجازی را نشان می‌دهد. جدول آدرس‌های نیمه چپ را به نیمه راست نگاشت می‌دهد. در این‌جا دو صفحه چهار مگابایتی به یک بخش چهار مگابایتی از حافظه فیزیکی نگاشت شده‌اند. یعنی برنامه می‌تواند با استفاده از دو آدرس به یک محتوا دسترسی یابد. این یکی دیگر از قابلیت‌های صفحه‌بندی است. در ادامه اجرا قرار است هسته تنها از بخش بالایی فضای آدرس مجازی استفاده نماید.<sup>33</sup> به عبارت دیگر، نگاشت پایینی حذف خواهد شد. علت اصلی این است که باید حافظه مورد دسترسی توسط هسته از دسترسی برنامه‌های کاربردی یا به عبارت دقیق‌تر برنامه‌های سطح کاربر<sup>34</sup> حفظ گردد. این یک شرط لازم برای ارائه سرویس امن به برنامه‌های سطح کاربر است. هر کد در حال اجرا دارای یک سطح دسترسی جاری<sup>35</sup> (`CPL`)

<sup>30</sup> Paging

<sup>31</sup> به طور دقیق‌تر این جداول سلسله‌مراتبی بوده و آدرس اولین لایه جدول در `cr3` قرار داده می‌شود.

<sup>32</sup> Page

<sup>33</sup> در `xv6` از آدرس `0x80000000` به بعد مربوط به سطح هسته و آدرس‌های `0x0` تا این آدرس مربوط به سطح کاربر هستند.

<sup>34</sup> User Level Programs

<sup>35</sup> Current Privilege Level



است. سطح دسترسی در پردازنده‌های xv6 از صفر تا سه متغیر بوده که صفر و سه به ترتیب ممتازترین و پایین‌ترین سطح دسترسی هستند. در سیستم عامل xv6 اگر  $CPL=0$  باشد در هسته و اگر  $CPL=3$  باشد در سطح کاربر هستیم<sup>36</sup>. تشخیص سطح دسترسی کد کنونی مستلزم خواندن مقدار ثبات CS است.<sup>37</sup> دسترسی به آدرس‌های هسته با  $CPL=3$  نباید امکان‌پذیر باشد. به منظور حفاظت از حافظه هسته، در مدخل جدول نگاشت‌های صفحه‌بندی، بیت‌هایی وجود دارد که حافظه هسته را از حافظه برنامه سطح کاربر تفکیک می‌نماید (پرچم  $PTE\_U$  (خط ۸۰۳) بیانگر حق دسترسی سطح کاربر به حافظه مجازی است). صفحه‌های بخش بالایی به هسته تخصیص داده شده و بیت مربوطه نیز این مسئله را تثبیت خواهد نمود. سپس توسط سازوکاری از دسترسی به مدخل‌هایی که مربوط به هسته هستند، زمانی که برنامه سطح کاربر این دسترسی را صورت می‌دهد، جلوگیری خواهد شد. در اینجا اساس تفکر این است که هسته عنصر قابل اعتماد سیستم بوده و برنامه‌های سطح کاربر، پتانسیل مخرب بودن را دارند.

20. به این ترتیب، در انتهای `entry.S`، امکان اجرای کد C هسته فراهم می‌شود تا در انتها تابع `main()` صدا زده (خط ۱۰۶۵) شود. این تابع عملیات آماده‌سازی اجزای هسته را بر عهده دارد. در مورد هر تابع به طور مختصر توضیح دهید. تابع معادل در هسته لینوکس را بیابید.

در کد `entry.S` هدف این بود که حداقل امکانات لازم جهت اجرای کد اصلی هسته فراهم گردد. به همین علت، تنها بخشی از هسته نگاشت داده شد. لذا در تابع `main()` تابع `kvmalloc()` فراخوانی می‌گردد (خط ۱۲۲۰) تا آدرس‌های مجازی هسته به طور کامل نگاشت داده شوند. در این نگاشت جدید، اندازه هر تکه پیوسته، ۴ کیلوبایت است. آدرسی که باید در `cr3` بارگذاری گردد، در متغیر `kpgdir` ذخیره شده است (خط ۱۸۴۲).

21. مختصری راجع به محتوای فضای آدرس مجازی هسته توضیح دهید.

22. علاوه بر صفحه‌بندی در حد ابتدایی از قطع‌بندی به منظور حفاظت هسته استفاده خواهد شد. این عملیات توسط `seginet()` انجام می‌گردد. همان‌طور که ذکر شد، ترجمه قطعه تأثیری بر ترجمه آدرس منطقی نمی‌گذارد. زیرا تمامی قطعه‌ها اعم از کد و داده روی یکدیگر می‌افتند. با این حال برای کد و داده‌های سطح کاربر پرچم `SEG\_USER` تنظیم شده است. چرا؟ (راهنمایی: علت مربوط به ماهیت دستورالعمل‌ها و نه آدرس است).

## اجرای نخستین برنامه سطح کاربر

تا به این لحظه از اجرا فضای آدرس حافظه هسته آماده شده است. بخش زیادی از مابقی تابع `main()`، زیرسیستم‌های مختلف هسته را فعال می‌نماید. مدیریت برنامه‌های سطح کاربر مستلزم ارائه انتزاعاتی برای ایجاد تمایز میان این برنامه‌ها و برنامه مدیریت آن‌ها است. کدی که تاکنون اجرا می‌شد را می‌توان برنامه مدیریت‌کننده سیستم و برنامه‌های سطح کاربر دانست.

23. جهت نگهداری اطلاعات مدیریتی برنامه‌های سطح کاربر ساختاری تحت عنوان `struct proc` (خط ۲۳۳۶) ارائه شده است. اجزای آن را توضیح داده و ساختار معادل آن در سیستم عامل لینوکس را بیابید.

از جمله اجزای ساختار `proc` متغیر `kpgdir` است که آدرس جدول مربوط به هر برنامه سطح کاربر را نگهداری می‌کند. مشاهده می‌شود که این آدرس با آدرس مربوط به جدول کد مدیریت‌کننده سیستم که در `kpgdir` برای کل سیستم نگهداری شده بود، متفاوت است. تا پیش از فراخوانی `userinit()` (خط ۱۲۳۵) تقریباً تمامی زیرسیستم‌های هسته فعال شده‌اند. جهت ارائه واسطی با کاربر از طریق ترمینال و همچنین آماده‌سازی بخش‌هایی از هسته که ممکن است توأم با به خواب رفتن کد باشد، تابع `userinit()` فراخوانی می‌گردد. این تابع وظیفه ایجاد نخستین برنامه سطح کاربر را دارد. ابتدا توسط تابع `allocproc()` برای این برنامه یک ساختار

<sup>36</sup> دو سطح دسترسی دیگر در اغلب سیستم عامل‌ها بلااستفاده است.

<sup>37</sup> در واقع در مد محافظت شده، دو بیت از این ثبات، سطح دسترسی کنونی را معین می‌کند. بیت‌های دیگر کاربردهای دیگری مانند تعیین افسر مربوط به قطعه در `gdt` دارند.

proc تخصیص داده می شود (خط ۲۵۲۵). این تابع بخش هایی را که برنامه برای اجرا در سطح ممتاز (هسته) نیاز دارد، مقداردهی می کند. یکی از عملیات مهمی که در این تابع صورت می گیرد، مقداردهی `p->context->eip` به آدرس تابع `forkret()` است. این عمل منجر به این می شود که هنگام اجرای برنامه<sup>38</sup> ابتدا `forkret()` اجرا گردد. آماده سازی بخش های باقی مانده سیستم در این تابع انجام می شود.

24. چرا به خواب رفتن در کد مدیریت کننده سیستم مشکل ساز است؟ (راهنمایی: به زمان بندی در ادامه توجه نمایید.)

در ادامه تابع `userinit()`، تابع `setupkvm()` فراخوانی شده و فضای آدرس مجازی هسته را برای برنامه سطح کاربر مقداردهی می کند.

25. تفاوت این فضای آدرس هسته با فضای آدرس هسته که توسط `kvmalloc()` در خط ۱۲۲۰ صورت گرفت چیست؟ چرا وضعیت به این شکل است؟

تابع `inituvm()` فضای آدرس مجازی سطح کاربر را برای این برنامه مقداردهی می نماید. به طوری که در آدرس صفر تا ۴ کیلوبایت، کد مربوط به `initcode.S` قرار گیرد.

26. تفاوت این فضای آدرس کاربر با فضای آدرس کاربر در کد مدیریت سیستم چیست؟

یک برنامه سطح کاربر می تواند برای دسترسی به سرویس های ممتاز سیستم به مد ممتاز ( $CPL=3$ ) منتقل شود. به این ترتیب می تواند حتی به حافظه هسته نیز دسترسی داشته باشد. به منظور تغییر مد امن، سازوکارهایی مانند فراخوانی سیستمی<sup>39</sup> وجود دارد. تفاوت در این سبک دسترسی این است که هسته آن را با یک سازوکار امن مدیریت می نماید. اجرای کد از فضای آدرس مجازی سطح کاربر به فضای آدرس مجازی هسته منتقل می شود. لذا باید وضعیت اجرای برنامه سطح کاربر در فضای آدرس مجازی سطح کاربر در مکانی ذخیره گردد. این مکان قاب تله<sup>40</sup> نام داشته و در ساختار `proc` ذخیره می شود.<sup>41</sup>

با توجه به این که اجرا در مد هسته است و جهت اجرای برنامه سطح کاربر باید به مد سطح کاربر منتقل شد، حالت سیستم به گونه ای شبیه سازی می شود که گویی برنامه سطح کاربر در حال اجرا بوده و تله ای رخ داده است. لذا فیلد مربوطه در `proc` باید مقداردهی شود. با توجه به این که قرار است کد به سطح کاربر بازگردد، بیت های مربوط به سطح دسترسی جاری ثبات های قطعه `p->tf->cs` و `p->tf->ds` به `DPL_USER` مقداردهی شده اند. `p->tf->eip` برابر صفر شده است (خط ۲۵۳۹). این بدان معنی است که زمانی که کد به سطح کاربر بازگشت، از آدرس مجازی صفر شروع به اجرا می کند. به عبارت دیگر اجرا از ابتدای کد `initcode.S` انجام خواهد شد. در انتها `p->state` به `RUNNABLE` مقداردهی می شود (خط ۲۵۵۰). این یعنی برنامه سطح کاربر قادر به اجرا است. حالت های ممکن دیگر یک برنامه در فصل زمان بندی بررسی خواهد شد.

در انتهای تابع `main()` تابع `mpmain()` فراخوانی شده (خط ۱۲۳۶) و به دنبال آن تابع `scheduler()` فراخوانی می شود (خط ۱۲۵۷). به طور ساده، وظیفه زمان بند تعیین شیوه اجرای برنامه ها بر روی پردازنده می باشد. زمان بند با بررسی لیست برنامه ها یک برنامه را که `p->state` آن `RUNNABLE` است بر اساس معیاری انتخاب نموده و آن را به عنوان کد جاری بر روی پردازنده اجرا می کند. این البته مستلزم تغییراتی در وضعیت جاری سیستم جهت قرارگیری حالت برنامه جدید (مثلاً تغییر `cr3` برای اشاره به جدول نگاشت برنامه جدید) روی پردازنده است. این تغییرات در فصل زمان بندی تشریح می شود. با توجه به این که تنها برنامه قابل اجرا برنامه `initcode.S` است، پس از مهیا شدن حالت پردازنده و حافظه در اثر زمان بندی، این برنامه اجرا شده و به کمک یک فراخوانی سیستمی برنامه `init.c` را اجرا نموده که آن برنامه نیز در نهایت یک برنامه ترمینال (خط ۸۵۲۹) را ایجاد می کند. به این ترتیب امکان ارتباط با سیستم عامل را فراهم می آورد.

27. کدام بخش از آماده سازی سیستم، بین تمامی هسته های پردازنده مشترک و کدام بخش اختصاصی است؟ (از هر کدام یک مورد را با ذکر دلیل توضیح دهید.) زمان بند روی کدام هسته اجرا می شود؟

<sup>38</sup> دقت شود اجرا هنوز در کد مدیریت کننده سیستم است.

<sup>39</sup> System Call

<sup>40</sup> Trap Frame

<sup>41</sup> تله لزوماً هنگام انتقال از مد کاربر به هسته رخ نمی دهد.

## 28. برنامه معادل initcode.S در هسته لینوکس چیست؟

## اشکال زدایی

کد هر برنامه‌ای ممکن است دارای اشکال باشد. اشکال زدایی ممکن است ایستا، پویا و یا به صورت ترکیبی صورت پذیرد. کشف اشکال در روش‌های ایستا، بدون اجرا و تنها بر اساس اطلاعات کد برنامه صورت می‌گیرد. به عنوان مثال کامپایلر Clang دارای تحلیل‌گرهای ایستا برای اشکال زدایی اشکال‌های خاص است. اشکال زدایی پویا که معمولاً دقیق‌تر است، اقدام به کشف اشکال در حین اجرای برنامه می‌نماید. ابزار leak-check در ابزار Valgrind یک اشکال‌زدای پویا برای تشخیص نشتی حافظه<sup>42</sup> است. از یک منظر می‌توان اشکال‌زداهای پویا را به دو دسته تقسیم نمود: (۱) اشکال‌زداهایی که بر یک نوع اشکال خاص مانند نشتی تمرکز دارند و (۲) اشکال‌زداهایی که مستقل از نوع اشکال بوده و تنها اجرا را ردگیری<sup>43</sup> نموده و اطلاعاتی از حالت سیستم (شامل سخت‌افزار و نرم‌افزار) در حین اجرا یا پس از اجرا جهت درک بهتر رفتار برنامه برمی‌گردانند. در این بخش ابزار اشکال‌زدای گنو<sup>44</sup> (GDB)، که یک اشکال‌زدای پویا از نوع دوم است معرفی خواهد شد.

GDB یک اشکال‌زدای متداول در سیستم‌های یونیکسی بوده که در بسیاری از شرایط، نقش قابل‌توجهی در تسریع روند اشکال‌زدایی ایفا می‌کند. اشکال‌زدایی برنامه‌های تک‌ریسه‌ای<sup>45</sup>، چندریسه‌ای<sup>46</sup> و حتی هسته‌های سیستم عامل توسط این ابزار ممکن است. جهت اشکال‌زدایی xv6 با GDB، در گام نخست باید سیستم عامل به صورتی بوت شود که قابلیت اتصال اشکال‌زدا به آن وجود داشته باشد. مراحل اتصال عبارت است از:

1. در یک ترمینال دستور `make qemu-gdb` اجرا گردد.
2. سپس در ترمینالی دیگر، فایل کد اجرایی به عنوان ورودی به GDB داده شود.
- چنانچه پیش‌تر ذکر شد کد اجرایی شامل یک نیمه هسته و یک نیمه سطح کاربر بوده که نیمه هسته، ثابت و نیمه سطح کاربر، بسته به برنامه در حال اجرا بر روی پردازنده دائماً در حال تغییر است. به این ترتیب، به عنوان مثال، هنگام اجرای برنامه cat، کدهای اجرایی سیستم شامل کد هسته و کد برنامه cat خواهند بود. جهت اشکال‌زدایی بخش سطح کاربر، کافی است دستور `gdb_cat` و جهت اشکال‌زدایی بخش هسته دستور `gdb` kernel فراخوانی شود. دقت شود در هر دو حالت، هر دو کد سطح هسته و کاربر اجرا می‌شوند. اما اشکال‌زدا فقط روی یک کد اجرایی (سطح کاربر یا هسته) کنترل داشته و تنها قادر به انجام عملیات بر روی آن قسمت خواهد بود.
3. نهایتاً با وارد کردن دستور `target remote tcp::26000` در GDB، اتصال به سیستم عامل صورت خواهد گرفت.

## روند اجرای GDB

GDB می‌تواند در هر گام از اجرا، با ارائه حالت سیستم، به برنامه‌نویس کمک کند تا حالت خطا را از حالت مورد انتظار تشخیص دهد. هنگام اجرای کد در GDB ممکن است چندین حالت رخ دهد:

1. اجرا با موفقیت جریان داشته باشد یا خاتمه یابد.
2. اجرا به علت اشکال، ناتمام مانده و برنامه متوقف شود.
3. اجرا متوقف نشده ولی حالت سیستم در برخی نقاط درونی یا در خروجی‌های برنامه نادرست باشد.

Memory Leak<sup>42</sup>Tracing<sup>43</sup>GNU Debugger<sup>44</sup>Single-Thread<sup>45</sup>Multithread<sup>46</sup>

هدف، یافتن حالات خطای سیستم در دو وضعیت ۲ و ۳ است. به عبارتی ابتدا باید در نقطه مورد نظر، توقف صورت گرفته و سپس به کمک دستورهای حالت سیستم را استخراج نمود. برای توقف اجرا در نقاط مختلف اجرا در GDB سازوکارهای مختلفی وجود دارد:

1. در اجرای ناتمام، اجرای برنامه به طور خودکار متوقف می‌شود.
2. با فشردن کلید ترکیبی Ctrl + C به اشکال‌زدا بازگشت.
- این عملیات در میان اجرا، آن را متوقف نموده و کنترل را به خط فرمان اشکال‌زدا منتقل می‌کند. مثلاً حلقه بی‌نهایت رخ داده باشد، می‌توان با این کلید ترکیبی، در نقطه‌ای از حلقه متوقف شد.
3. روی نقطه‌ای از برنامه Breakpoint قرار داد. بدین ترتیب هر رسیدن اجرا به این نقطه منجر به توقف اجرا گردد.
- روش‌های مختلفی برای تعیین نقطه استقرار Breakpoint وجود داشته که در این [لینک قابل](#) مشاهده است.

از جمله:

#### انتخاب نام و شماره خط فایل

\$ break cat.c:12

#### انتخاب نام تابع

\$ b cat

#### انتخاب آدرس حافظه

\$ b \*0x98

این نقاط می‌توانند در سطح کاربر یا هسته سیستم عامل باشند. همچنین می‌توانند شرطی تعریف شوند.

4. روی خانه خاصی از حافظه Watchpoint قرار داد تا دسترسی یا تغییر مقدار آن خانه، منجر به توقف اجرا گردد.

Watchpoint ها انواع مختلفی داشته و با دستورهای خاص خود مشخص می‌گردند.

دستور زیر:

\$ watch \*0x1234567

یک Watchpoint روی آدرس 0x1234567 در حافظه می‌گذارد. بدین ترتیب نوشتن در این آدرس، منجر به توقف اجرا خواهد شد.

می‌توان از نام متغیر هم استفاده نمود. مثلاً Watch v، watch را روی (آدرس) متغیر v قرار می‌دهد. باید دقت نمود، اگر Watch روی متغیر محلی قرار داده شود، با خروج از حوزه دسترسی به آن متغیر، Watch حذف شده و به برنامه‌نویس اطلاع داده می‌شود. اگر هم آدرسی از فضای پشته<sup>47</sup> داده شود، ممکن است در حین اجرا متغیرها یا داده‌های نامرتب دیگر در آن آدرس نوشته شود. یعنی این آدرس در زمان‌های مختلف مربوط به داده‌های مختلف بوده و در عمل Watch کارایی مورد نظر را نداشته باشد. یک مزیت مهم Watch، تشخیص وضعیت مسابقه<sup>49</sup> است که در فصول بعدی درس با آن آشنا خواهید شد. در این شرایط می‌توان تشخیص داد که کدام ریس<sup>50</sup> یا پردازنده مقدار نامناسب را در آدرس حافظه نوشته که منجر به خطا شده است.

همان‌طور که مشاهده می‌شود، خیلی از حالات با استفاده از چهار سازوکار مذکور به سهولت قابل استخراج نیستند. مثلاً حالتی که یک زنجیره خاص فراخوانی توابع وجود داشته باشد یا این که مثلاً حالتی خاص در داده‌ساختارها رخ داده و یک لیست پیوندی، چهارمین عنصرش را حذف نماید.

- ۱) برای مشاهده Breakpoint ها از چه دستوری استفاده می‌شود؟
- ۲) برای حذف یک Breakpoint از چه دستوری و چگونه استفاده می‌شود؟

<sup>47</sup> Stack

<sup>48</sup> یعنی فضای آدرسی که داده‌هایی از جمله مقادیر متغیرهای محلی و آدرس‌های برگشت مربوط به توابع فراخوانی شده در آن قرار دارد.

<sup>49</sup> Race Condition

<sup>50</sup> Thread

## کنترل روند اجرا و دسترسی به حالت سیستم

پس از توقف می‌توان با استفاده از دستورهای به حالت سیستم دسترسی پیدا نمود. همچنین دستورهای برای تعیین شیوه ادامه اجرا وجود دارد. در ادامه، برخی از دستورهای کنترلی و دسترسی به حالت اجرا معرفی خواهد شد.

پس از توقف روی Breakpoint می‌توان با اجرای دستورهای step و next و finish به ترتیب به دستور بعدی، به درون دستور بعدی (اگر فراخوانی تابع باشد) و به خارج از تابع کنونی (یعنی بازگشت به تابع فراخواننده) منتقل شد. به عبارت دیگر، اجرا گام‌به‌گام قابل بررسی است. بدین معنی که پیش از اجرای خط جاری برنامه سطح کاربر یا هسته، امکان دستیابی به اطلاعات متغیرها و ثبات‌ها فراهم می‌باشد. به این ترتیب می‌توان برنامه را از جهت وجود حالات نادرست، بررسی نمود. همچنین دستور continue اجرا را تا رسیدن به نقطه توقف بعدی یا اتمام برنامه ادامه می‌دهد.

(۳) دستور زیر را اجرا کنید. خروجی آن چه چیزی را نشان می‌دهد؟

```
$ bt
```

(۴) دو تفاوت دستورهای x و print را توضیح دهید. چگونه می‌توان محتوای یک ثبات خاص را چاپ کرد؟ (راهنمایی: می‌توانید از دستور help استفاده نمایید: help x و help print با دستور list می‌توان کد نقطه توقف را مشاهده نمود.)

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

(۶) به کمک استفاده از GDB، درباره ساختار struct input موارد زیر را توضیح دهید:

- توضیح کلی این struct و متغیرهای درونی آن و نقش آن‌ها
- نحوه و زمان تغییر مقدار متغیرهای درونی (برای مثال، input.e در چه حالتی تغییر می‌کند و چه مقداری می‌گیرد)

## اشکال زدایی در سطح کد اسمبلی

اشکال زدایی برنامه در سطوح مختلفی قابل انجام است. با توجه به این که بسیاری از جزئیات اجرا در کد سطح بالا (زبان سی<sup>51</sup>) قابل مشاهده نیست، نیاز به اشکال زدایی در سطح کد اسمبلی خواهد بود. به عنوان مثال بهینه‌سازی‌های ممکن است ترتیب اجرا در کد سطح بالا را تغییر داده یا بخشی از کد را حذف نماید. به عنوان مثال دیگر می‌توان از شیوه دسترسی به جداول لینکر نام برد. جزئیات دسترسی به یک تابع کتابخانه‌ای خاص یا یک متغیر سراسری آن کتابخانه دسترسی شده است، در سطح کد اسمبلی و با دسترسی به جداول لینک رخ داده و در سطح زبان سی قابل رؤیت نیست.

با فشردن همزمان سه دکمه Ctrl + X + A رابط کاربری متنی<sup>52</sup> GDB یا همان TUI گشوده شده و کد اسمبلی مربوط به نقطه توقف، قابل رؤیت است. برای اطلاعات بیشتر در رابطه با این رابط کاربری می‌توانید به این [صفحه](#) مراجعه کنید.

(۷) خروجی دستورهای layout src و layout asm در TUI چیست؟

(۸) برای جابجایی میان توابع زنجیره فراخوانی جاری (نقطه توقف) از چه دستورهایی استفاده می‌شود؟ دستورهای stepi و nexti معادل‌های سطح اسمبلی step و next بوده و به جای یک دستور سی، در ریزدانی یک دستور العمل ماشین عمل می‌کنند. در شرایطی که کد مورد اشکال زدایی از ابتدا در زبان اسمبلی نوشته شده باشد، چاره‌ای جز استفاده از این دستورها وجود نخواهد داشت.

## نکات پایانی

با توجه به کاستی‌هایی که در اشکال‌زدها وجود دارد، همچنان برخی از تکنیک‌ها در کدزنی می‌تواند بسیار راهگشا باشد. ساده‌ترین راه برای اشکال‌زدایی این است که تغییرها را اندک انجام داده و گام‌به‌گام از صحت

<sup>51</sup> C

<sup>52</sup> Text user interface

اجرای کد، اطمینان حاصل شود. به عنوان مثال اگر آرایه‌ای ۱۰۰ عنصری تخصیص داده شده و در نقطه‌ای فراتر از مرز انتهایی آن نوشتن صورت گیرد، حافظه‌ای غیر از حافظه مربوط به آرایه دستکاری می‌گردد. چندین حالت ممکن است رخ دهد. از جمله اینکه:

1. اقدام به نوشتن در حافظه‌ای فقط خواندنی مانند کد برنامه، صورت پذیرد. در چنین شرایطی خطا رخ داده و نقطه توقف به راحتی در GDB قابل رؤیت خواهد بود.
2. در حافظه نوشتنی نامرتب نوشته شده و مشکلی پیش نیاید.
3. در حافظه نوشتنی نامرتب نوشته شود و اجرای برنامه به طرز عجیبی متوقف گردد. به طوری که GDB نقطه نامربوطی را نشان دهد. یعنی تأثیر آن بلافاصله و به طور مستقیم رخ ندهد. در چنین شرایطی استفاده ابتدایی از اشکال‌زدا راحتی راهگشا خواهد بود. چک کردن اندازه آرایه و احتمال دسترسی به خارج آن در سطح کد، می‌توانست راحت‌تر باشد. البته در برخی موارد به سادگی و یا با تکنیک‌هایی مانند استفاده از Watch، ضبط اجرا و حرکت رو به عقب از حالت نادرست، می‌توان اشکال را یافت<sup>53</sup>. اما تکنیک قبلی بهتر بود.

بنابراین، استفاده از GDB در کنار دیگر ابزارها و تکنیک‌ها در پروژه‌های این درس توصیه می‌گردد. با توجه به آشنایی اولیه‌ای که با GDB فراهم شده است، می‌توان مزایای آن را برشمرد:

- اشکال‌زدایی کدهای بزرگ و کدهایی که با پیاده‌سازی آن‌ها آشنایی وجود ندارد. ممکن است نیاز باشد یک کد بزرگ را به برنامه اضافه کنید. در این شرایط اشکال‌زدایی اجرای Crash کرده در GDB درک اولیه‌ای از نقطه خرابی ارائه می‌دهد.
- بررسی مقادیر حالت برنامه، بدون نیاز به قرار دادن دستورهای چاپ مقادیر در کد و کامپایل مجدد آن.
- بررسی مقادیر حالت سخت‌افزار و برنامه که در سطح کد قابل رؤیت نیستند. به عنوان مثال مقدار یک اشاره‌گر به تابع، مقصد یک تابع کتابخانه‌ای، اطمینان از قرارگیری آدرس متغیر محلی در بازه حافظه پشته، این که اجرا در کدام فایل کد منبع قرار دارد، اطلاع از وضعیت فضای آدرس حین اجرا، مثلاً این که هر کتابخانه در چه آدرسی بوده و در کدام کتابخانه در حال اجرا هستیم و ....
- تشخیص اشکال‌های پیچیده مانند این که کدام ریس، یک متغیر را دستکاری نموده یا چرا یک متغیر مقدار نادرستی داشته یا مقداردهی اولیه نشده است. این اشکال‌های با کمک Watch و ضبط و اجرای مجدد رو به جلو/عقب به راحتی قابل تشخیص هستند.

<sup>53</sup> GDB در برنامه‌های عادی قادر به ضبط و اجرای رو به عقب برنامه است. همچنین ابزار RR که توسط شرکت موزیلا برای اشکال‌زدایی فایرفاکس ارائه شده است امکان انجام همین عملیات را به صورت قطعی دارد. این قطعیت، در اشکال‌زدایی کدهای همروند و وضعیت مسابقه بسیار کمک‌کننده است.

## نکات مهم

- برای تحویل پروژه ابتدا یک مخزن خصوصی در سایت GitHub ایجاد نموده و سپس پروژه خود را در آن Push کنید. سپس اکانت UT-OS-TA را با دسترسی Maintainer به مخزن خود اضافه نمایید. کافی است در محل بارگذاری در سایت درس، آدرس مخزن، شناسه آخرین Commit و گزارش پروژه را بارگذاری نمایید.
- به سؤالاتی که در صورت پروژه از شما خواسته شده است پاسخ دهید و آن‌ها را در گزارش کار خود بیاورید.
- همه اعضای گروه باید به پروژه آپلود شده توسط گروه خود مسلط باشند و لزوماً نمره افراد یک گروه با یکدیگر برابر نیست.
- در صورت مشاهده هرگونه شباهت بین کدها یا گزارش دو گروه، نمره 0 به هر دو گروه تعلق می‌گیرد.
- سؤالات را در کوتاه‌ترین اندازه ممکن پاسخ دهید.