به نام خدا

درس سیستم‌های عامل

نیم‌سال دوم ۹۹-۹۸

دانشکدهٔ مهندسی کامپیوتر

دانشگاه صنعتی شریف

مدرس **مهدی خرازی**

تمرین گروهی **یک**

موضوع **برنامه‌های کاربر**

موعد تحویل مستند طراحی **ساعت ۲۳:۵۹ دوشنبه ۱۲ اسفند ۱۳۹۸**

موعد تحویل کد و گزارش نهایی **ساعت ۲۳:۵۹ دوشنبه ۲۶ اسفند ۱۳۹۸**

فهرست مطالب

به اولین تمرین از تمرین‌های گروهی درس خوش آمدید :)
در این درس از سیستم عامل آموزشی **pintos** استفاده می‌کنیم. دلیل استفاده از این سیستم عامل خاص این است که در قالب توسعه دادن یک هسته‌ی [۱] سیستم عامل واقعی -که به درستی کار می‌کند و در عین حال پیچیدگی‌های اضافی ندارد- توسط خودتان، به شما درکی عینی از مفاهیم اصلی یک سیستم عامل داده شود. شالوده‌ی [۲] سیستم عامل **pintos** ، ناتوانی زیادی در سامان دادن به پرونده‌ها [۳] ، زمان‌بندی بین ریسه‌ها [۴] و اجرای برنامه‌های کاربر [۵] دارد. شما در پروژه‌های این درس، این شالوده را توسعه خواهید داد و ویژگی‌های مذکور را به آن اضافه خواهید کرد.
این مستند، برای بیان کردن جزییات تمرین گروهی اول است. به منظور شفافیت مطالب، جزییات تمرین به طور مجزا در قسمت «وظیفه‌ی شما» آمده است. اطلاعات کامل‌تر در سند مربوط به قسمت «منابع» آمده است و این قسمت در حین طراحی و پیاده‌سازی راه‌حل‌هایی که برای بخش‌های مختلف تمرین دارید، مفید خواهد بود. احتمالا بهتر باشد که با مطالعه‌ی قسمت «منابع» شروع کنید تا یک دید کلی نسبت به سیستم عامل **pintos** به دست آورید و پس از آن، تلاش کنید که جزییاتی را که در قسمت «وظیفه‌ی شما» ذکر شده است، متوجه شوید.

۱ راه‌اندازی

۱.۱ دریافت شالوده‌ی کد pintos

در ماشین مجازی دستورهای زیر را اجرا کنید:

```
cd /home/vagrant/code/handouts
git pull
```

۲.۱ راه‌اندازی مخزن تمرین‌های گروهی

در ماشین مجازی خود به مسیر زیر بروید:

```
cd /home/vagrant/code/
```

سپس فرمان زیر را اجرا کنید:

```
git clone https://tarasht.ce.sharif.ir/ce424-982-groups/ce424-982-groupX ./group
```

که به جای **X** ، باید شماره‌ی گروه خود را قرار دهید.
پرونده‌های مربوط به **pintos** را که در مسیر **handouts** قرار دارند به مسیر **group** منتقل کنید و توسعه‌ی کد خود را در همان مسیر **group** انجام دهید و **push** کنید.

۲ وظیفه‌ی شما

شما در این پروژه، توانایی اجرا کردن برنامه‌های کاربر را به هسته‌ی سیستم عامل خواهید افزود. کدی که به شما به عنوان شالوده داده شده است، توانایی بار کردن [۶] یک برنامه در حافظه را دارد اما برنامه‌ها قادر نیستند که آرگومانهای خط فرمان [۷] را بخوانند یا فراخوانی‌های سیستمی [۸] انجام دهند (مثلا از یک پرونده یا سوکت شبکه چیزی بخوانند یا درون آن چیزی بنویسند).

[۱] Kernel
[۲] Skeleton
[۳] File System
[۴] Thread Scheduling
[۵] User Programs
[۶] Load
[۷] Command Line
[۸] System Call (Syscall)

۱.۲ ماموریت اول: پاس دادن آرگومان‌های خط فرمان به برنامه

در **pintos** ، از تابع **process_execute(char *file_name)** برای ایجاد پردازه‌های جدید در سطح کاربر استفاده می‌شود. در حال حاضر این تابع از آرگومان‌های خط فرمان پشتیبانی نمی‌کند. وظیفه‌ی شما این است که این توانایی را به آن بیفزایید. بعد از افزودن این قابلیت توسط شما، مثلا با صدا زدن این تابع به شکل **process_execute("ls -ahl")** ، دو آرگومان به شکل **["ls", "-ahl"]** ایجاد خواهد شد و از طریق متغیرهای **argv** و **argc** ، این آرگومان‌ها به برنامه پاس داده خواهند شد.

تعداد زیادی از برنامه‌های تست **pintos** ، در ابتدای اجرا، نام خودشان -یعنی **argv[0]**- را چاپ می‌کنند. از آنجایی که قابلیت پاس دادن آرگومان هنوز پیاده‌سازی نشده است، هر یک از این برنامه‌ها که تلاش کند که **argv[0]** را بخواند، از کار خواهد افتاد. بنابراین تا زمانی که قابلیت پاس دادن آرگومان‌ها را پیاده‌سازی نکرده‌اید، این برنامه‌ها کار نخواهند کرد و در این تست‌ها ناموفق خواهید بود.

۲.۲ ماموریت دوم: فراخوانی‌های سیستمی برای کنترل پردازه‌ها

در حال حاضر، **pintos** فقط فراخوانی سیستمی مربوط به **exit** را می‌شناسد و اجرا می‌کند. شما باید قابلیت پشتیبانی از فراخوانی‌های سیستمی **halt** ، **exec** ، **wait** و **practice** را اضافه کنید. تابع متناظر با هر یک از این فراخوانی‌های سیستمی در **pintos/src/lib/user/syscall.c** که کتابخانه‌ای در سطح کاربر است، وجود دارد. هر یک از این توابع، آرگومان‌های مربوط به فراخوانی سیستمی متناظر خود را آماده می‌کند و سپس انتقال به حالت کاری هسته [۹] را انجام می‌دهد (یعنی کنترل سیستم به دست هسته‌ی سیستم عامل سپرده می‌شود). پس از انتقال به حالت هسته، مسئولیت رسیدگی به فراخوانی‌های سیستمی در هسته‌ی سیستم عامل، با توابعی است که در کتابخانه‌ی **pintos/src/userprog/syscall.c** وجود دارند.

فراخوانی سیستمی **practice** ، کاری ساده انجام می‌دهد: آرگومان خود را با یک جمع می‌کند و نتیجه را برمی‌گرداند (هدف از این فراخوانی سیستمی، دست‌گرمی است تا بتوانید تابع‌های رسیدگی‌کننده به سایر فراخوانی‌ها را راحت‌تر پیاده‌سازی کنید). سیستم توسط فراخوانی سیستمی **halt** خاموش می‌شود.

فراخوانی سیستمی **exec** ، برنامه‌ای جدید را با استفاده از تابع **process_execute()** آغاز می‌کند. (دقت کنید که فراخوانی سیستمی **fork** در سیستم عامل **pintos** وجود ندارد. در واقع عملکرد فراخوانی سیستمی **exec** در **pintos** ، مشابه این است که در سیستم عامل **Linux** ابتدا **fork** را فراخوانی می‌کنید و سپس در پردازه‌ی فرزند، **execve** را فراخوانی کنید.) فراخوانی سیستمی **wait** برای اتمام یک پردازه‌ی فرزند مشخص صبر می‌کند.

در ابتدا برای این که بتوانید فراخوانی‌های سیستمی مذکور را پیاده‌سازی کنید، لازم است که بتوانید عملیات‌های خواندن و نوشتن در حافظه (مطابق با فضای آدرسی مجازی مربوط به پردازه‌ی کاربر) را به طور «ایمن و بی‌دردسر» انجام دهید. در واقع آرگومان‌های فراخوانی سیستمی در پشته‌ی [۱۰] پردازه‌ی کاربر -به طور دقیق، بالای جایی که اشاره‌گر مربوط به پشته [۱۱] به آنجا اشاره می‌کند- قرار دارند. هسته‌ی سیستم عامل شما نباید در صورت تلاش برای خواندن مقدار ذخیره شده در مقصدِ [۱۲] یک اشاره‌گر نامعتبر یا پوچ [۱۳] ، دچار از کار افتادگی شود. به طور مثال، فرض کنید که یک فراخوانی سیستمی انجام می‌شود و هسته‌ی سیستم عامل تلاش می‌کند که به کمک اشاره‌گر پشته، آرگومان‌های آن فراخوانی سیستمی را بخواند و فرض کنید که به دلایلی نامعلوم، اشاره‌گر پشته مقداری نامعتبر دارد. در چنین مثالی، هسته‌ی سیستم عامل نباید به علت تلاش کردن برای خواندن آرگومان‌های فراخوانی سیستمی از پشته، دچار از کارافتادگی شود. همچنین بعضی از آرگومان‌های فراخوانی سیستمی، اشاره‌گرهایی به بافرها یا رشته‌های درون فضای آدرس پردازه‌ی کاربر هستند و این اشاره‌گرها هم می‌توانند مقادیری نامعتبر یا پوچ داشته باشند!

بنابراین لازم است که شما همه‌ی حالت‌هایی را که یک فراخوانی سیستمی به دلیل خطاهای حافظه‌ای به اتمام نمی‌رسد، در نظر بگیرید و به شکلی ایمن و بی‌دردسر، آنها را از سر بگذرانید! این خطاهای حافظه‌ای شامل این حالت‌ها است: اشاره‌گرهای پوچ، اشاره‌گرهای نامعتبر (که به جاهایی از حافظه اشاره می‌کنند که نگاشته [۱۴] نشده‌اند) و اشاره‌گرهایی که به فضای آدرس مجازی متعلق به هسته‌ی سیستم عامل اشاره می‌کنند. توجه کنید که در دسترسی‌های ۴بایتی به حافظه (مانند **32-bit integer**) ممکن است که ۲بایت آن معتبر و ۲بایت آن نامعتبر باشد. این اتفاق زمانی می‌افتد که حافظه میان دو صفحه در حافظه قرار می‌گیرد. در این حالات شما باید برنامه‌ی کاربر را خاتمه دهید. پیشنهاد می‌شود این قسمت از کد خود را قبل از پیاده‌سازی فراخوانی‌های سیستمی پیاده‌سازی و تست نمایید. (برای اطلاعات بیش‌تر می‌توانید به قسمت **Accessing User Memory** در سند منابع مراجعه نمایید.)

---

[۹] Kernel Mode
[۱۰] Stack
[۱۱] Stack Pointer
[۱۲] Dereference
[۱۳] Null
[۱۴] Mapped

۳.۲ ماموریت سوم فراخوانی‌های سیستمی برای عملیات روی پرونده‌ها

افزون بر فراخوانی‌هایی که مربوط به کنترل پردازه‌ها هستند، شما بایستی فراخوانی‌های سیستمی **open** ، **remove** ، **create** ، **filesize** ، **read** ، **write** ، **seek** ، **tell** و **close** را برای پرونده‌ها پیاده‌سازی کنید. **pintos** در حال حاضر یک سامانه‌ی مدیریت پرونده‌ی ساده و ابتدایی دارد. کافی است که پیاده‌سازی شما برای فراخوانی‌های سیستمی مذکور، توابع موجود در کتابخانه‌ی مربوط به **file system** را به شکلی مناسب فراخوانی کند. در واقع شما نیاز ندارید که هیچ یک از این عملیات‌ها را خودتان پیاده‌سازی کنید.

سامانه‌ی مدیریت پرونده‌ای که **pintos** دارد، **thread-safe** نیست. شما بایستی اطمینان حاصل کنید که پیاده‌سازی شما برای فراخوانی‌های سیستمی برای عملیات روی پرونده‌ها، به طور همزمان چندین تابع از سامانه‌ی مدیریت پرونده را فراخوانی نکند. در تمرین گروهی ۳، روش‌های همگام‌سازی [۱۵] نسبتا پیچیده‌تری به سامانه‌ی مدیریت پرونده‌ی **pintos** اضافه خواهید کرد ولی فعلا در این تمرین گروهی، شما اجازه دارید که از یک **global lock** برای اعمال مربوط به سامانه‌ی مدیریت پرونده‌ها استفاده کنید. در واقع می‌توان با این **global lock** این طور انگاشت که کل قطعه‌کد مربوط به **file system** در تمرین شما، یک ناحیه بحرانی [۱۶] است و بدین ترتیب می‌توان **thread safety** را تضمین کرد. توصیه می‌شود که در این تمرین، کلا محتوای درون **filesys/** را تغییر ندهید.

شما بایستی اطمینان حاصل کنید که در حین اجرای یک پردازه‌ی کاربر، کسی نمی‌تواند پرونده اجرایی آن پردازه را تغییر دهد. تست‌های **rox** اطمینان حاصل می‌کنند که شما مانع از نوشتن روی پرونده‌های اجرایی مربوط به یک پردازه‌ی در حال اجرا می‌شوید. توابع **file_deny_write()** و **file_allow_write()** برای ایجاد این قابلیت به شما کمک می‌کنند. ممانعت از نوشتن روی پرونده‌های اجرایی مربوط به پردازه‌های در حال اجرا، امری مهم است زیرا ممکن است که یک سیستم عامل، **page** های مربوط به کد را به کندی از پرونده بخواند و در حافظه بگذارد یا چند **page** از **page** های مربوط به کد را از حافظه پاک کند و بعدا آنها را از پرونده مجددا بخواند. در سیستم عامل **pintos** ، این امر، امری حیاتی نیست زیرا **pintos** قبل از اجرای هر پرونده، کل آن را درون حافظه بار می‌کند و سپس آن را اجرا می‌کند و همچنین **pintos** از هیچ نوعی از **demand paging** پشتیبانی نمی‌کند. به هر حال، از آنجایی که افزودن این قابلیت تمرین خوبی است، شما همچنان ملزم به پیاده‌سازی این قابلیت هستید.

نکته: کد نهایی شما در تمرین گروهی اول، به عنوان نقطه‌ی شروع شما در تمرین گروهی سوم استفاده خواهد شد. بعضی از تست‌های مربوط به تمرین گروهی سوم، به بعضی از فراخوانی‌هایی سیستمی که در این تمرین پیاده‌سازی می‌کنید، بستگی دارد و ممکن است که نیاز به تغییر دادن پیاده‌سازی خود در بعضی از این فراخوانی‌های سیستمی پیدا کنید تا از قابلیت‌هایی که در تمرین گروهی سوم لازم می‌شود، پشتیبانی کنید. بنابراین بایستی که در حین طراحی کردن پیاده‌سازی خود در این تمرین، این نکته را در ذهن خود داشته باشید.

۳ تحویل‌دادنی‌ها

نمره‌ی تمرین گروهی شما از سه مولفه‌ی زیر تشکیل شده است:

- ۲۰ درصد برای مستند طراحی و جلسه‌ی مرور طراحی
- ۶۵ درصد کد و پیاده‌سازی
- ۱۵ درصد برای گزارش نهایی و کیفیت کد

۱.۳ مستند طراحی (مهلت تحویل تا پایان ۱۲ اسفند) و جلسه‌ی مرور طراحی

قبل از این که شروع به کد زدن کنید، بایستی برای پیاده‌سازی خود یک نقشه‌ی راه داشته باشید و بدانید که قصد دارید هر ویژگی را چطور پیاده‌سازی کنید و همچنین باید بتوانید خودتان را قانع کنید که طراحی را به درستی انجام داده‌اید و اشکالی در آن نیست. برای این تمرین گروهی، بایستی که یک مستند طراحی تحویل بدهید و در جلسه‌ی مرور طراحی، شرکت کنید. در این جلسه، دستیاران آموزشی با شما در مورد طراحی مد نظر شما مشورت خواهند کرد و از شما سوالاتی خواهند پرسید و بایستی بتوانید از طراحی خود دفاع کنید.

قالب مستند طراحی این پروژه در آدرس **doc/project1.md** قرار دارد. شما باید این مستند را کامل نمایید و در همان آدرس قراردهید. مستند طراحی در فرمت **Markdown** است. برای مشاهده آن می‌توانید در طرشت به آدرس پرونده بروید و آن را در قالب نهایی مشاهده نمایید.

سه قسمت در مستند طراحی **pintos** وجود دارد. اولین قسمت آشنایی با **pintos** است. در این قسمت شما باید مرحله به مرحله پیش بروید و جواب همه سوالات را در در مستند طراحی بنویسید. قسمت دوم بررسی اجمالی طراحی است. شما باید طراحی

[۱۵] Synchronization
[۱۶] Critical Section

خودتان را که قصد دارید پروژه را با آن انجام دهید، به صورت اجمالی توضیح دهید. قسمت سوم سوالاتی است که افزون بر طراحی پرسیده شده‌است. در ادامه هر قسمت از مستند طراحی با جزییات توضیح داده‌شده‌است.

۱.۱.۳ آشنایی با pintos

طبیعت پروژه **pintos** به این صورت است که فرد برای طراحی و پیاده‌سازی یک جواب خوب نیاز به آشنایی به کدهای آن دارد. هدف این تمرین کمک برای آشنایی اولیه شما با کدهای **pintos** است.

۱.۱.۱.۳ یافتن دستور معیوب

در ابتدا، دستورات **make** و **make check** را در پوشه‌ی **pintos/src/userprog** اجرا نمایید. مشاهده می‌نمایید که هم‌اکنون هیچ تستی به طور موفقیت‌آمیز اجرا نمی‌شود. ما قدم به قدم تست **do-nothing** را در **GDB** اجرا می‌نماییم تا در **pintos** تغییری ایجاد نماییم که در نتیجه‌ی این تغییر، این تست با موفقیت اجرا شود. همچنین، متوجه بشویم که درحال حاضر، سیستم عامل **pintos** پشتیبانی از برنامه‌های کاربر [۱۷] را به چه ترتیبی پیاده‌سازی کرده است.

ما از **do-nothing** استفاده می‌کنیم زیرا این تست، ساده‌ترین تست برای آزمودن توانایی **pintos** در پشتیبانی از برنامه‌های کاربر است. شما باید پرونده‌ی **pintos/src/tests/userprog/do-nothing.c** را مطالعه نمایید.این پرونده، کد یک برنامه است که هیچ کاری انجام نمی‌دهد. تنها جمله‌ی تابع **main** ، **return 162** است به این معنا که برنامه کد خروج ۱۶۲ را به سیستم عامل می‌دهد. عدد ۱۶۲ به خودی خود بی اهمیت است. ما عدد ۱۶۲ را به جای ۰ انتخاب کرده‌ایم تا دنبال کردن نحوه اداره کردن این مقدار توسط هسته‌ی **pintos** راحت‌تر باشد **(162 = 0xa2)** .

زمانی که شما **make** را اجرا کردید، **do-nothing.c** به برنامه اجرایی **do-nothing** کامپایل شده است. این پرونده در آدرس **pintos/src/userprog/build/tests/userprog/do-nothing** قرار دارد. تست **do-nothing** برنامه اجرایی **do-nothing** را در **pintos** با کمک دستور **pintos run** اجرا می‌کند (بخش **Runnning Pintos** را در قسمت منابع مشاهده نمایید).

محتوای داخل **pintos/src/userprog/build/tests/userprog/do-nothing.result** را ببینید. این پرونده نشان‌دهنده‌ی خروجی تست **do-nothing** است که توسط چارچوب آزمونگرِ [۱۸] **pintos** اجرا شده است. چارچوب آزمونگر انتظار دارد که خروجی تست **"do-nothing: exit(162)"** باشد. این پیام استانداردی است که **pintos** در زمانی که یک پردازه خارج می‌شود چاپ می‌کند. اما همانطور که در **diff** نشان داده شد، **pintos** این خروجی را نمی‌دهد. به جای آن، **do-nothing** در **userspace** به دلیل دسترسی غیر مجاز به حافظه **(Segmantation Fault)** ، دچار **crash** می‌شود. با توجه به محتوای **do-nothing.result** به سوالات زیر پاسخ دهید:

۱. برنامه سعی کرد به چه آدرسی از آدرسهای مجازی حافظه دسترسی پیدا کند که باعث **crash** شد؟

۲. آدرس مجازی دستوری که باعث **crash** شد چیست؟

۳. پرونده اجرایی **do-nothing** را توسط **objdump** (با این ابزار در تمرین ۰ آشنا شدید) واهم‌گذاری [۱۹] کنید. نام تابعی که برنامه در آن **crash** کرد، چیست؟ دستوری که باعث **crash** می‌شود چیست؟

۴. کد C تابعی که در بالا نامش را یافتید پیدا نمایید. (راهنمایی: کد در فضای کاربر [۲۰] اجرا شده است، پس کد در **do-nothing.c** یا در یکی از پرونده‌های در دو پوشه‌ی **pintos/src/lib** و **pintos/src/lib/user** است.) در مورد هر دستوری که در قسمت قبلی واهم‌گذاری کرده‌اید، به طور مختصر توضیح دهید که چرا لازم هستند. (راهنمایی: **80x86 Calling Convention** را ببینید.)

۵. چرا دستوری که در قسمت ۳ شناسایی کردید سعی کرد به آدرسی که در قست ۱ شناسایی کردید، دسترسی یابد؟ جواب را با توجه به مقدار ثبات‌ها [۲۱] ندهید، بلکه سعی کنید جوابی سطح بالاتر و کلی‌تر بدهید.

۲.۱.۱.۳ به سوی Crash

حال که فهمیدیم چرا **do-nothing** دچار **crash** می‌شود، اجرای تست **do-nothing** در **pintos** را از زمان **boot**

[۱۷] User Program
[۱۸] Testing Framework
[۱۹] Disassemble
[۲۰] User space
[۲۱] Registers

شدن سیستم دنبال می‌کنیم. هدف ما این است که متوجه شویم چگونه **userprogram loader** را تغییر دهیم تا تست **do-nothing** ، **crash** نکند. همچنین با نحوه پشتیبانی **pintos** از برنامه‌های کاربر آشنا شویم. برای این کار مسیر کاری خودتان را به **pintos/src/userprog/** تغییر دهید و دستور زیر را اجرا نمایید:

```
pintos --gdb --filesys-size=2 -p ./build/tests/userprog/do-nothing -a do-nothing -- -q -f
    run do-nothing
```

در یک **terminal** دیگر به مسیر **pintos/src/userprog/build** بروید. برنامه‌ی **GDB** اجرا کنید **(pintos-gdb)** **./kernel.o** . و آن را به پردازه‌ی **pintos** متصل نمایید **(debugpintos)** . اگر قسمتی نامفهوم است به بخش **Debugging** **Pintos** و **Debugging Pintos Tests** در قسمت منابع مراجعه نمایید.

زمانی که دستور **debugpintos** را وارد می‌نمایید، پردازنده هنوز شروع به کار نکرده‌است. به صورت سطح‌بالا موارد ذیل قبل از این که **pintos** پردازه **do-nothing** را اجرا نماید باید اتفاق بیافتد.

- ابتدا **BIOS** ، **bootloader** مربوط به **pintos** **(pintos/src/threads/loader.S)** را از اولین سکتور دیسک می‌خواند و در آدرس **0x7c00** می‌نویسد.
- سپس **bootloader** کد هسته‌ی **pintos** را از دیسک می‌خواند و در آدرس **0x20000** می‌نویسد. و سپس به نقطه‌ی شروعِ کدِ هسته **(pintos/src/threads/start.S)** پرش می‌کند.
- کد در نقطه‌ی شروع هسته حالت پردازنده را به **32-bit protected mode** [۲۲] تغییر می‌دهد و تابع **main()** را صدا می‌زند **(pintos/src/threads/init.c)** .
- تابع **main()** ، **pintos** را آماده به کار می‌کند. برای این کار زمانبند[۲۳] ، زیرسیستم حافظه [۲۴] ، بردار وقفه [۲۵] ، دستگاه‌های سخت‌افزاری [۲۶] و سامانه‌ی مدیریت پرونده‌ها [۲۷] را آماده‌سازی می‌نماید.

یک **breakpoint** بر روی **run_task** قرار دهید و در **GDB** ادامه دهید تا تنظیمات را رد نمایید. همانطور که در کد **run_task** می‌توانید ببینید، **pintos** برنامه **do-nothing** (برنامه‌ای که در دستور ورودی مشخص کرده بودیم) را با فراخواندن **pro-cess_wait (process_execute ("do-nothing"));** از **run_task()** اجرا می‌نماید. هر دو تابع **process_wait** و **process_execute** در پرونده **pintos/src/userprog/process.c** قرار دارند.

به سوالات زیر پاسخ دهید:

۶. در **GDB** به داخل تابع **process_execute** بروید. نام و آدرس ریسه‌ای که این تابع را اجرا می‌نماید چیست؟ چه ریسه‌های دیگری در این زمان در **pintos** وجود دارند؟ ساختار **threads** مربوط به آن‌ها را کپی نمایید. (راهنمایی: برای قسمت آخر دستور **dumplist &all_list thread allelem** ممکن است کارآمد باشد.)

۷. **backtrace** برای ریسه کنونی چیست؟ **backtrace** را به عنوان جواب از **GDB** کپی نمایید. همچنین کد **C** که مربوط به فراخوانی هر تابع هست را نیز در جواب قرار دهید.

۸. یک **breakpoint** بر روی **start_process** قرار دهید و ادامه دهید تا به این تابع برسید. چه ریسه‌های دیگری در این زمان در **pintos** وجود دارند؟ ساختار **threads** مربوط به آن‌ها را کپی نمایید و در جواب قرار دهید.

۹. در کجا ریسه‌ای که **start_process** را اجرا می‌کند ساخته شده است؟ خط‌های کد را کپی نمایید و در جواب قرار دهید.

۱۰. در **GDB** به داخل تابع **start_process** بروید قدم به قدم پیش بروید تا جایی که به فراخوانی تابع **load()** برسید. **load()** مقادیر **eip** و **esp** را در ساختار **if_** تنظیم می‌کند. مقادیر ساختار **if_** را به در مبنا ۱۶ چاپ نمایید (راهنمایی: **print/x if_**).

۱۱. اولین دستور در **intr_exit** **asm volatile** اشاره‌گر پشته را به پایین ساختار **if_** تنظیم می‌کند. دستور بعدی به پرش می‌کند. در داخل کد توضیحات بیشتری قرار دارد. در **GDB** به داخل **asm volatile** بروید و قدم به قدم دستورات را اجرا نمایید تا به **iret** برسید.مشاهده می‌کنید که به **userspace** برمی‌گردید. چرا حالت پردازنده در زمان اجرای این تابع تغییر کرد؟

---

[۲۲] https://en.wikipedia.org/wiki/Protected_mode
[۲۳] Scheduler
[۲۴] Memory Subsystem
[۲۵] Interrupt Vector
[۲۶] Hardware Device
[۲۷] File System

۱۲. بعد از اجرای **iret** ، دستور **info registers** را وارد نمایید تا محتویات رجیسترها را مشاهده نمایید. تفاوت این مقدارها با مقدارهای داخل ساختار **_if** چیست؟

۱۳. حال اگر بخواهید از **backtrace** استفاده نمایید، متوجه می‌شوید که تنها یک آدرس در مبنای ۱۶ دریافت می‌نمایید. زیرا **pintos-gdb ./kernel.o** فقط نمادهای هسته را می‌خواند. حال که در **userspace** هستیم، باید نمادها را باید از پرونده اجرایی‌ای که درحال حاضر **pintos** آن را اجرا می‌کند بخوانیم. الان **pintos** در حال اجرای **do-nothing** است. با دستور **loadusersymbols tests/userprog/do-nothing** این نمادها را بارگذاری می‌نماییم. اگر دستور **backtrace** را وارد نمایید مشاهده می‌کنید که در تابع **_start** هستید. با دستورات **disassemble** و **stepi** قدم به قدم دستورات را اجرا نمایید تا **pagefault** اتفاق بیافتد. در این لحظه پردازشگر وارد حالت هسته می‌شود تا به **page fault** رسیدگی نماید. اگر در زمان **page fault** دستور **backtrace** را بزنید دیگر پشته‌ی کاربر را نمی‌بینید، بلکه پشته‌ی هسته را می‌بینید. با این حال با دستور **btpagefault** می‌توانید پشته‌ی کاربر را ببینید. محتوای **btpagefault** را کپی نمایید.

۳.۱.۱.۳ Debug

دستورالعمل معیوبی که در قسمت ۳ یافتید باید با دستوری که در **GDB** مشاهده نمودید، مطابقت داشته باشد. حال که دستور معیوب را پیدا کرده‌اید، هدف آن را فهمیده‌اید و قدم به قدم فهمید که برنامه به چه شکل توسط هسته اجرا می‌شود، باید کد هسته را طوری ویرایش نمایید که تست **do-nothing** به درستی اجرا شود.

۱۴. هسته **pintos** را طوری تغییر دهید که **do-nothing** دیگر **crash** نکند. تغییرات شما باید در هسته باشد، نه در برنامه‌ی **userspace** (**do-nothing.c**) یا کتابخانه‌های در **pintos/src/lib**. این تغییرات نباید زیاد باشند. این کار با یک خط نیز امکان پذیر است. بعد از انجام این تغییر تست **do-nothing** باید قبول شود و بقیه‌ی تست‌ها رد. تغییراتی را که ایجاد کردید و دلیل لزوم آن را بیان نمایید.

۱۵. ممکن است تغییراتی که ایجاد نمودید باعث قبولی تست **do-stack-align** نیز بشوند. یک نگاه به تست **do-stack-align** بیندازید. مشابه تست **do-nothing** است ولی مقدار خروجی‌اش، **$esp % 16** است. مقداری که این برنامه باید برگرداند را بنویسید (راهنمایی: می‌توانید جواب را در **do-stack-align.ck** بیابید.) و توضیح دهید چرا این مقدار را برمی‌گرداند. حال در صورتی که تغییراتتان باعث قبولی این تست نشده بود، آن‌ها را تغییر دهید.

۱۶. **GDB** را دوباره اجرا نمایید. دستور **loadusersymbols** را نیز اجرا نمایید. یک **breakpoint** بر روی **_start** قرار دهید و اجرا را ادامه دهید تا به آن برسید. با استفاده از **disassemble** و **stepi** اجرا را ادامه دهید تا به **int $0x30** در پرونده **pintos/src/lib/user/syscall.c** برسید. در این نقطه ۲ کلمه‌ی بالای پشته را چاپ نمایید (راهنمایی: **x/2xw $esp**) و خروجی را کپی نمایید.

۱۷. دستور **int $0x30** پردازشگر را به حالت هسته می‌برد و یک قاب بردار وقفه در پشته‌ی هسته قرار می‌دهد. قدم به قدم به پیش بروید تا به **syscall_handler** برسید. مقادیر **args[0]** و **args[1]** چیست؟ این دو چگونه به جواب قست قبل مربوط اند؟

۱۸. به داخل **thread_exit()** بروید و سپس به داخل **process_exit()** بروید. دلیل وجود سمافور **temporary** چیست؟ همانطور که می‌بینید، تابع **process_exit()** ، **sema_up (&temporary);** را صدا می‌زند. **sema_down** متناظر با آن در کجا قرار دارد؟

۱۹. یک **breakpoint** بر روی **sema_down** که مکان آن را در قسمت قبلی یافتید، قرار دهید و ادامه دهید تا به آن برسید. اگر به درستی این کار را انجام داده باشید باید به **breakpoint** ای که قرار داده‌بودید، برسید. نام و آدرس ریسه‌ای که این تابع را اجرا می‌نماید چیست؟ چه ریسه‌های دیگری در این زمان در **pintos** وجود دارند؟

حال اگر ادامه دهید، بعد از پایان اجرای **do-nothing** ، **pintos** اقدام به خاموش شدن می‌نماید چون به هنگام شروع آن را با گزینه‌ی **-q** اجرا نمودیم. اگر درباره روش خاموش شدن **pintos** کنجکاوید، می‌توانید در **GDB** تا انتها قدم به قدم پیش‌بروید.

تبریک! شما اجرای یک برنامه کاربر را در **pintos** از اول تا آخر مشاهده‌نمودید. امیدواریم این سوالات باعث آشنایی شما با **pintos** و کد آن شده‌باشد.

۲.۱.۳ بررسی اجمالی طراحی

برای هریک از ۳ بخش پروژه شما باید طراحی خود را از چهار جنبه (که در ادامه آورده می‌شود) توضیح دهید. پیشنهاد می‌کنیم که برای هر بخش از پروژه یک بخش در سند خود ایجاد کنید و آن را به ۴ زیر بخش تقسیم کنید و هر جنبه را جداگانه توضیح دهید.

۱. داده‌ساختارها و توابع: هر داده ساختار، متغیرهای **global** و یا **static** و **typedef** ها و یا **enum** هایی که به کد اضافه کردید یا تغییر دادید را به صورت کد **C** (نه شبه کد) بیان کنید. همراه کدها توضیح حداقلی در مورد هدف این تکه کد دهید (توضیحات مفصل‌تر در بخش‌های بعدی مطلوبست).

۲. الگوریتم‌ها: در این بخش توضیح می‌دهید که چرا کد شما کار می‌کند! توضیحات شما باید مفصل‌تر از توضیحاتی باشد که در سند تمرین آمده است (سند تمرین موجود است و تکرار آن بی‌مورد است).از طرفی در نظر داشته باشید که توضیحات این بخش باید از سطح کد بالاتر باشد و لازم نیست که خط به خط کد توضیح داده شود صرفاًباید ما را قانع کنید که کد شما نیازمندی مطرح شده را برطرف کرده است. لازم به ذکر است که در این جا باید شرایطی که استثنا و حالت خاص به حساب می‌آیند توضیح داده شوند. انتظار می‌رود که هنگام نوشتن سند طراحی کد **pintos** را مطالعه کرده باشید و در صورت لزوم، ارجاعاتی به **pintos** داشته باشید.

۳. به هنگام‌سازی [۲۸]: در این بخش شما باید تمامی منابعی که بین ریسه‌های مختلف مشترک هستند را بیان کنید. و برای هریک بگویید که چگونه به این منبع دسترسی وجود دارد (به عنوان مثال از یک وقفه مشخص) و پس از آن استراتژی دسترسی و تغییر دادن امن آن منبع را بیان کنید. برای هر منبع نشان دهید که طراحی شما نحوه دسترسی صحیح را ایجاد می‌کند و از به وجود آمدن بن‌بست [۲۹] جلوگیری می‌کند. به صورت کلی، بهترین استراتژی‌های به‌هنگام‌سازی ساده هستند و به سادگی قابل اعتبارسنجی‌اند. اگر توضیح‌دادن استراتژی‌هایتان دشوار است، نشانگر این است که باید روش‌های ساده‌تری در پیش بگیرید. لازم است که روش خودتان را از نظر هزینه زمانی و حافظه‌ای بررسی کنید و نشان دهید آیا روش شما محدودیت شدیدی بر هم‌رَوی [۳۰] کارهای هسته و/یا برنامه‌های کاربر اعمال کرده‌است یا خیر. وقتی در مورد موازاتی [۳۱] که توسط روش شما میسر شده است بحث می‌کنید، توضیح دهید که ریسه‌ها با چه توالی و فرکانسی به منبع مشترک دسترسی پیدا می‌کنند و آیا محدودیتی در تعداد ریسه‌هایی که می‌خواهند به ناحیه‌های بحرانی مستقل در یک زمان دسترسی پیدا کنند وجود دارد یا خیر. هدف شما باید دوری از روش‌های قفل کردن منابع باشد.

۴. منطق: توضیح دهید چرا طراحی شما از دیگر روش‌هایی که بررسی کردید بهتر است و نارسایی‌های آن را شرح دهید. به نکته‌هایی مثل اینکه چقدر طراحی قابل درک است، چقدر برنامه‌نویسی آن زمان‌بر است و پیچیدگی الگوریتم‌های شما از نظر زمانی و حافظه چقدر است و این که طراحی شما تا چه حد برای افزودن ویژگی‌های جدید انعطاف‌پذیر است، توجه داشته باشید.

۳.۱.۳ سوال‌های افزون بر طراحی

۱. تست‌های این تمرین گروهی در **pintos/src/tests/userprog** قرار دارد. در یکی از آن‌ها یک اشاره‌گر نامعتبر به عنوان ورودی فراخوانی سیستمی داده شده است تا تست شود آیا پیاده‌سازی شما به صورت مناسب امکان خواندن و نوشتن در فضای حافظه‌ی کاربر را فراهم می‌کند یا خیر. تستی را پیدا کنید که از یک اشاره‌گر به پشته‌ی (**%esp**) نامعتبر در فراخوانی سیستمی استفاده می‌کند. نام و شماره‌ی خط کد را ذکر کنید و به طور دقیق بیان کنید که هدف این تست چیست.

۲. تستی را پیدا کنید که از یک اشاره‌گر به پشته‌ی (**%esp**) معتبر هنگام فراخواندن یک فراخوانی سیستمی استفاده می‌کند اما به دلیل نزدیکی اشاره‌گر به مرز بین فضای حافظه‌ی کاربر و فضای حافظه‌ی هسته، برخی از ورودی‌های فراخوانی سیستمی در محدوده‌ی غیرمجاز قرار می‌گیرند. نام، شماره خط و هدف تست را به صورت دقیق توضیح دهید.

۳. یک بخش از پروژه را شناسایی نمایید که توسط تست‌های موجود به صورت کامل پوشش داده نمی‌شود. توضیح دهید برای تست کردن آن بخش باید چه تست‌هایی اضافه شود (بیش از یک جواب صحیح وجود دارد).

۴.۱.۳ بازخورد طراحی

شما در یک جلسه ی ۲۰-۲۵ دقیقه ای، طراحی خود را به دستیار آموزشی پروژه ارائه می‌دهید. در آن جلسه باید آماده باشید تا به سوالات دستیار آموزشی در مورد طراحی خود پاسخ دهید و از طراحی خود دفاع کنید.

۵.۱.۳ نمره دهی

مستند طراحی و بازخورد طراحی با هم نمره‌دهی می‌شوند. این بخش ۲۰ نمره دارد که بر اساس توضیحات شما از طراحی در مستند طراحی و پاسخ‌دهی به سوالات در جلسه ی بازخورد طراحی نمره‌دهی می‌شود. باید حتما در جلسه بازخورد طراحی حضور داشته باشید تا نمره‌ای به شما تعلق گیرد.

[۲۸] Synchronization
[۲۹] Deadlock
[۳۰] Concurrency
[۳۱] Parallelism

۲.۳ پیاده‌سازی

نمره‌ی پیاده‌سازی شما توسط نمره‌دهنده ی خودکار داده می‌شود. **pintos** یک مجموعه تست دارد که می‌توانید خودتان آن را اجرا کنید. دقیقا همین تست‌ها برای نمره‌دهی شما استفاده می‌گردد لازم به ذکر است با تغییر دادن تست‌ها تغییری در تست‌هایی که سامانه‌داوری اجرا می‌کند ایجاد نمی‌شود و نمره‌ای که از آن بدست می‌آید ملاک است.پس از هربار نمره‌دهی، نمره دهنده‌ی خودکار نمره شما را در پرونده **grade.txt** قرار می‌دهد.

۳.۳ پیشنهادات

پیشنهاد می‌کنیم که پروژه را با پیاده‌سازی فراخوانی سیستمی **write** برای توصیف کننده‌ی پرونده [۳۲] خروجی استاندارد **(STDOUT)** شروع کنید. پس از اتمام پیاده‌سازی این قابلیت تست **stack-align-1** باید پاس شود.

پس از تمام کردن کار بالا دستور **printf()** برای فضای کاربر قابل استفاده است حال فراخوانی سیستمی **practice** و بخش پاس دادن آرگومان‌ها را کامل کنید.

در گام بعد مطمئن شوید که پیام **exit(-1)** چاپ شود حتی اگر یک برنامه به دلیل یک خطا از اجرا خارج شد. در حال حاضر زمانی چاپ می‌شود که فراخوانی سیستمی **exit** از فضای کاربر فراخوانی شود و اگر برنامه طلب دسترسی به فضایی غیرمجاز دسترسی‌پیداکند، **exit code** چاپ نمی‌شود.

۴.۳ گزارش نهایی

نمره‌دهی گزارش شما بر مبنای دو چیز است: **اول**، بایستی برای هر **commit** ، پیام دقیقی نوشته باشید. بدین منظور پس از مشخص کردن پرونده‌هایی که قصد دارید آنها را **commit** کنید، فرمان زیر را اجرا کنید.

```
git commit
```

بعد از این فرمان، برای شما ویرایشگری باز خواهد شد که در آن پیام خود را بنویسید. پیام شما باید به گونه‌ای شفاف باشد که هم‌گروهی شما با خواندن فقط همین پیام، متوجه وضعیت کنونی پروژه شود. تلاش کنید طوری این پیام‌ها را بنویسید که حتی بدون نیاز به دیدار حضوری با یکدیگر، کار گروهی خود را انجام دهید و هماهنگ بمانید (البته که می‌توانید حضوری هم کار کنید! ولی ما فرض می‌کنیم که هر کدام در قاره‌ای متفاوت قرار دارید! :)).

برای نمونه، می‌توانید اسلوب نوشتن چنین پیامهایی را در **changelog** های هسته‌ی سیستم عامل **Linux** ببینید. بدیهی است که انتظار نوشتن پیامهایی به این تفصیل وجود ندارد اما پیام شما باید حداقل اطلاعات زیر را داشته باشد:

```
Add some feature/Fix some bugs(some should be explained)

Test 27 passed but test 28 and 31 that related to that feature has some issues.
In line ... of file ... this pointer has invalid value that caused that problem(that
    should be explained)
```

به طور خاص، بایستی دقیق بودن پیامهای خود را هنگام تلفیق کردن انشعابهای غیراصلی در انشعاب **master** رعایت کنید.

دوم ،بعد از اتمام کد پروژه باید یک گزارش از پیاده‌سازی خود آماده کنید. گزارش خود را در مسیر **reports/project1.md** قرار دهید.موارد زیر در گزارش شما مطلوب است:

- تغییراتی که نسبت به سند طراحی اولیه داشتید و دلیلی که این تغییر را انجام دادید را بیان کنید(در صورت لزوم آوردن بحث‌های خود با دستیار آموزشی مانعی ندارد)
- بیان کنید که هر فرد گروه دقیقا چه بخشی را انجام داد؟ آیا این کار را به صورت مناسب انجام دادید و چه کارهایی برای بهبود عملکرد تان می‌توانید انجام دهید.

کد شما بر اساس کیفیت کد نیز نمره دهی خواهد شد. موارد بررسی از این دست می‌باشند:

- آیا کد شما مشکل بزرگی امنیتی در بخش حافظه دارد (به صورت خاص رشته‌ها در زبان **C**)؟ **memory leak** و نحوه مدیریت ضعیف خطاها نیز بررسی خواهد شد.
- آیا از یک **Code Style** واحد استفاده کردید؟ آیا **style** مورد استفاده توسط شما با **pintos** هم‌خوانی دارد؟(از نظر فرورفتگی و نحوه نام‌گذاری)

[۳۲] File Descriptor

- آیا کد شما ساده و قابل درک است؟
- آیا کد پیچیده‌ای در بخشی از کدهای خود دارید؟ در صورت وجود آیا با قرار دادن توضیحات مناسب آن را قابل فهم کردید؟
- آیا کد **Comment** شده‌ای در کد نهایی خود دارید؟
- آیا کدی دارید که کپی کرده باشید؟
- آیا الگوریتم‌های **linked list** را خودتان پیاده‌سازی کردید یا از پیاده‌سازی موجود استفاده کردید؟
- آیا طول خط کدهای شما بیش از حد زیاد است؟ (۱۰۰ کاراکتر)
- آیا در **git** خودتان پرونده‌های **binary** حضور دارند؟(پرونده‌های **binary** و پرونده‌های **log** را **commit** نکنید!)

۴ منابع

منابع مورد نیاز شما در دو سند وجود دارد که پیوند آنها در زیر آمده است:

۱. **Project1**

۲. **Pregame**

لطفا بخش ۴ (منابع) سند **Project1** را کامل مطالعه کنید. مطالبی که در بخش ۴ (منابع) در سند **Pregame** وجود دارند، بسیار مفید هستند و به شما کمک خواهند کرد؛ اما خواندن آنها اجباری نیست و در صورتی که نیاز به اطلاعات بیشتری (مثلا در مورد **GDB** و ...) داشتید می‌توانید به بخش منابع این سند مراجعه کنید.

۵ سوالات پرتکرار

۱. چه مقدار کد باید بزنیم؟
میزان تغییرات پاسخ ما (با استفاده از برنامه **diffstat**) در ادامه آماده است. خط آخر نشان‌دهنده تعداد همه خط‌هایی است که اضافه یا حذف شده اند. دقت کنید که خطوطی که تغییر کرده اند هم به عنوان خط حذف شده و هم به عنوان خط اضافه شده محاسبه می‌شوند.
پاسخ‌های فراوان دیگری نیز وجود دارند که ممکن است تغییرات فراوانی نسبت به پاسخ ما داشته باشند و ممکن است فایل‌هایی که تغییر داده اند با فایل‌هایی که ما تغییر داده ایم متفاوت باشند.(فایل‌هایی را تغییر داده باشند که ما تغییر نداده ایم یا فایل‌هایی که ما تغییر داده ایم را تغییر نداده باشند.)

```
threads/thread.c | 13
threads/thread.h | 26 +
userprog/exception.c | 8
userprog/process.c | 247 +++++++++++++++--
userprog/syscall.c | 468 ++++++++++++++++++++++++++++++++-
userprog/syscall.h | 1
6 files changed, 725 insertions(+), 38 deletions(-)
```

۲. وقتی **pintos -p file -- -q** را اجرا می‌کنم، **kernel panic** رخ می‌دهد.
آیا سیستم را با دستور **pintos -f** فرمت کردید؟
آیا اسم فایل بیش از حد طولانی است؟ فایل سیستم محدودیت ۱۴ حرفی برای نام فایل‌ها دارد. دستوری مانند **pintos -p ../../examples/echo -- -q** این محدودیت را نقض می‌کند. برای مثال می‌توانید از دستور **pintos -p ../../examples/echo -a echo -- -q** استفاده کنید تا نام فایل را **echo** قرار دهد.
آیا فایل سیستم پر شده است؟
آیا فایل سیستم بیشتر از ۱۶ فایل دارد؟ محدودیت حداکثر ۱۶ فایل برای فایل سیستم پایه وجود دارد.
فایل سیستم می‌تواند بیش از حد تکه تکه شده باشد تا فضای پیوسته کافی برای فایل شما فراهم نباشد.

۳. وقتی دستور **pintos -p ../file -- ** را اجرا می‌کنم، فایل کپی نمی‌شود.
به صورت پیش‌فرض فایل‌ها با نامی که به آن اشاره می‌کنید نوشته می‌شوند، پس در این حالت نام فایل کپی شده **../file** خواهد بود. می‌توانید به جای این دستور از **pintos -p ../file -a file -- ** استفاده کنید. همچنین می‌توانید لیست فایل‌های موجود در فایل سیستم را با دستور **pintos -q ls** مشاهده کنید. فایل سیستم پایه **pintos** پوشه‌ها را پشتیبانی نمی‌کند.

۴. همه‌ی برنامه‌های کاربر با **page fault** قطع می‌شوند.
اگر پاس دادن آرگومان‌ها را پیاده سازی نکرده باشید یا به اشتباه پیاده سازی کرده باشید این مشکل پیش می‌آید. کتابخانه پایه C برای برنامه‌های کاربر تلاش می‌کند که **argc** و **argv** را بخواند، بنابراین اگر پشته به خوبی تنظیم نشده باشد نیز این مشکل رخ می‌دهد.

۵. همه‌ی برنامه‌های کاربر با فراخوانی سیستمی قطع می‌شوند.
شما باید برای دیدن هر چیزی در ابتدا فراخوانی‌های سیستمی را پیاده سازی کنید. هر برنامه‌ی معقولی حداقل فراخوانی سیستمی مربوط به خروج **(exit())** را صدا می‌زند. تابع **printf()** نیز **write()** را صدا می‌زند. رسیدگی کننده پیش‌فرض به فراخوانی سیستمی فقط عبارت **system call** چاپ می‌کند و به فراخوانی سیستمی **exit()** رسیدگی می‌کند. تا قبل از پیاده سازی فراخوانی‌های سیستمی می‌توانید از تابع **hex_dump()** برای بررسی درستی پیاده‌سازی پاس دادن آرگومان‌ها استفاده کنید.(می‌توانید برای توضیحات بیشتر به **Program Startup Details** در سند منابع مراجعه کنید.)

۶. چطور می‌توانم برنامه‌های کاربر را **disassemble** کنم؟
با استفاده از ابزار **obj-dump(80x86)** یا **i386-elf-objdump (SPARC)** می‌توانید این کار را انجام دهید. برای این کار از دستور **objdump -d <file>** استفاده کنید. برای یک تابع خاص نیز می‌توانید از دستور **disassemble** در **GDB** استفاده کنید.

۷. چرا بسیاری از فایل‌های **include** مربوط به C در برنامه‌های **pintos** کار نمی‌کند؟ آیا می‌توانم از **libFOLAN** استفاده کنم؟
کتابخانه **C** که فراهم شده است بسیار محدود است و شامل بسیاری از ویژگی‌هایی که از یک کتابخانه **C** در یک سیستم‌عامل واقعی انتظار می‌رود نمی‌شود. با توجه به اینکه کتابخانه‌های **C** باید فراخوانی‌های سیستمی برای **I/O** یا اختصاص حافظه را تولید کند، باید مخصوص سیستم‌عامل (و معماری) ساخته شوند.
اگر کتابخانه‌ای مانند قسمتی از کتابخانه‌های استاندارد **C** فراخوانی سیستمی انجام دهد، احتمال بسیار بالا با **pintos** کار نمی‌کند. **pintos** از رابطی غنی برای فراخوانی‌های سیستمی مانند سیستم‌عامل‌های واقعی مانند **Linux** یا **FreeBSD** پشتیبانی نمی‌کند،همچنین از شماره متفاوتی **(0x30)** برای قطع برنامه‌ها برای فراخوانی سیستمی نسبت به **Linux** استفاده می‌کند **(0x80)** .
احتمال زیادی دارد که کتابخانه‌ی مد نظر شما از قسمتهایی از کتابخانه‌ی **C** که **pintos** پیاده سازی نکرده، استفاده کرده باشد. بنابراین، احتمالا باید مقداری تلاش کنید تا آن را با **pintos** سازگار کنید. به‌طور ویژه به این نکته دقت کنید که کتابخانه‌ی برنامه‌های کاربر **C** در **pintos**، دستور **malloc()** را پیاده سازی نکرده است.

۸. چگونه برنامه‌های کاربر جدید را کامپایل کنم؟
فایل **src/examples/Makefile** را تغییر دهید و سپس دستور **make** را اجرا کنید.

۹. آیا می‌توانم برنامه‌های کاربر را تحت **debugger** اجرا کنم؟
با مقداری محدودیت می‌توان این کار را انجام داد. بخش مربوطه را مشاهده کنید.

۱۰. چه تفاوتی بین **tid_t** و **pid_t** وجود دارد؟
برای شناسایی ریسه‌های هسته (**kernel threads**) که روی آن می‌تواند یک پردازه مربوط به کاربر در حال اجرا باشد یعنی اگر با **process_execute()** ساخته شده باشد، یا با استفاده از **thread_create()** ساخته شده باشد، از **tid_t** استفاده می‌شود و فقط در هسته از آن استفاده می‌شود.
همچنین **pid_t** برای شناسایی پردازه‌های کاربر استفاده می‌شود که توسط پردازه‌های کاربر یا هسته با استفاده در فراخوانی‌های سیستمی **exec** یا **wait** استفاده می‌شود.
شما می‌توانید هر نوعی که می‌خواهید برای **tid_t** و **pid_t** انتخاب کنید. به طور پیش‌فرض هر دو آن‌ها **int** هستند.

شما می‌توانید از یک نگاشت یک به یک بین آن‌ها استفاده کنید تا یک مقدار برابر یک پردازه را نشان دهد یا می‌توانید از نگاشت‌های پیچیده تری استفاده کنید.

۱.۵ پاس دادن آرگومان‌ها

- آیا بالای پشته در حافظه مجازی هسته نیست؟
بالای پشته بر **PHYS_BASE** (معمولا **0xc0000000**) قرار دارد که همان‌جایی است که هسته شروع می‌شود. ولی قبل از این که پردازنده داده را درون پشته قرار دهد، اشاره‌گر پشته را یکی کم می‌کند. بنابراین اولین مقدار (که ۴ بایت است) در آدرس **0xbffffffc** قرار می‌گیرد.

- آیا **PHYS_BASE** ثابت است؟
خیر؛ شما باید حالاتی که **PHYS_BASE** یکی از مضارب **0x10000000** بین **0x80000000** تا **0xf0000000** باشد را به سادگی با کامپایل دوباره پشتیبانی کنید.

- چگونه باید به چندین فاصله در یک لیست آرگومان‌ها رسیدگی کرد؟
با چندین فاصله باید مانند یک فاصله رفتار شود. همچنین نیازی به پشتیبانی هیچ کاراکتر خاص دیگری به غیر از فاصله نیست.

- آیا می‌توانم یک کران بالا برای اندازه لیست آرگومان‌ها قرار دهم؟
بله می‌توانید یک کران بالای منطقی برای آن قرار دهید.

۲.۵ فراخوانی سیستمی

- آیا می‌توان از **struct file *** برای گرفتن توصیف‌کننده فایل استفاده کرد؟ آیا می‌توان از **struct thread *** برای **pid_t** استفاده کرد؟
شما باید خودتان این تصمیمات را برای طراحی بگیرید. بیشتر سیستم‌های عامل بین توصیف‌کننده فایل‌ها (یا شناسه پردازه‌ها) و آدرس ساختمان‌داده مربوط به آن‌ها در هسته تمایز قائل می‌شوند. قبل از پیاده‌سازی ممکن است بخواهید روی دلایل آن کمی فکر کنید.

- آیا می‌توان کران بالایی برای تعداد فایل‌های باز برای هر پردازه قرار داد؟
بهتر است که این‌کار را انجام ندهید ولی اگر ضروری است می‌توانید از کران ۱۲۸ فایل استفاده کنید.

- اگر فایلی که باز است حذف شود چه اتفاقی می‌افتد؟
شما باید برای فایل‌ها سیستم معنایی **Unix** را پیاده‌سازی کنید، به این صورت که وقتی فایلی حذف شد،همه پردازه‌هایی که توصیف‌کننده آن فایل را دارند باید بتوانند از آن استفاده کنند و به این معنی است که باید بتوانند در آن فایل بنویسند یا از آن بخوانند. فایل دیگر اسمی ندارد و پردازه‌های دیگر نمی‌توانند آن را باز کنند، ولی تا زمانی که همه توصیف‌کننده‌هایی که به آن فایل اشاره می‌کنند بسته نشده اند یا سیستم خاموش نشده فایل وجود دارد.

- چگونه می‌توان آن دسته از برنامه‌های کاربر را که به بیش از **4KB** فضای پشته نیاز دارند، اجرا کرد؟
می‌توانید کد تنظیمات برپایی پشته را تغییر دهید تا بیشتر از یک صفحه از فضای حافظه را به هر پردازه قرار دهد. برای این پروژه این لازم نیست.

- چه اتفاقی باید بیفتد اگر یک دستور **exec** در میانه‌ی کار خراب شود؟
اگر در هر صورتی پردازه‌ی فرزند بارگیری نشود باید **-1** برگرداند که شامل مشکل در بارگیری قسمتی از پردازه نیز هست.(مانند زمانی که در تست **multi-oom** حافظه کم بیاورد.) بنابراین پردازه‌ی پدر نباید قبل از مشخص شدن این که بارگیری موفق بوده یا

نه، از **exec** بازگردد. پردازه‌ی فرزند باید این اطلاعات را از طریق هماهنگ‌سازی مناسب(برای مثال از طریق سمافور) به پردازه‌ی پدر منتقل کند تا مطمئن باشد بدون رخ دادن **race condition** این اطلاعات منتقل شود.

۶ توصیه‌ها

۱.۶ توصیه‌های عمومی

شما باید قبل از شروع کار روی پروژه، کد منبع **pintos** که می خواهید اصلاح کنید را بخوانید و درک کنید. به همین دلیل است که از شما می‌خواهیم که مستند طراحی بنویسید و باید حداقل باید درک بالایی روی فایل‌هایی مانند **process.c** داشته باشید تا مشکلات مفهومی روی طراحی شما تاثیر نگذارد و زمان پیاده سازی مشکلی پیش نیاید.
شما باید یادبگیرید که از قابلیت‌های پیچیده **GDB** استفاده کنید. برای این پروژه دیباگ کردن کد معمولا از پیاده‌سازی آن بیشتر زمان می‌گیرد، هرچند فهم خوب از کدزی که تغییر می‌دهید می‌تواند به شما کمک کند که مشکلات می‌توانند مربوط به کجای آن باشند. تاکید می‌کنیم که فایل‌هایی که می‌خواهید تغییر دهید را حتما بخوانید و متوجه شوید. (با این هشدار که حجم کدها زیاد است و بیش از حد خود را درگیر نکنید.)
همچنین فایل‌های **binary** و فایل‌های **log** را **push** نکنید.
به این نکته نیز توجه داشته باشید که این پروژه وقت زیادی از شما خواهد گرفت.

۲.۶ کار گروهی

در گذشته ، بسیاری از گروه‌ها هر تکلیف را به صورت قطعاتی تقسیم می کردند و هرکس روی قطعه مربوط به خود کار می‌کرد. نزدیک ددلاین می‌خواستند که کدهای خود را ترکیب کنند و پاسخ خود را ارسال کنند، این کار خوبی نیست و آن را توصیه نمی‌کنیم. این گروه‌ها به این نتیجه می‌رسند که برخی تغییراتی که انجام داده اند با هم تعارض دارند و به زمان زیاد برای دیباگ کردن نیاز دارد. برخی از گروه‌هایی که از این روش استفاده کردند به کدهایی رسیدند که حتی کامپایل یا بوت هم نشدند و تعداد کمی از تست‌ها را پاس کردند.
به‌جای این روش توصیه می‌کنیم که با استفاده از **git** تغییرات خود را به مرور و سریع‌تر یکی کنید. با این روش در ادامه‌ی پروژه، با احتمال کمتری غافل‌گیر می‌شوید چون هر کس می‌تواند کد دیگران را در همان زمانی که نوشته می‌شوند، مشاهده کنند. همچنین این قابلیت وجود دارد که تغییرات را مرور کنید و اگر مشکلی رخ داد به نسخه‌ای که کار می‌کند برگردید.
همچنین توصیه می‌کنیم که به صورت گروهی کد بزنید. زیرا در این صورت احتمال رخ دادن مشکل در کد کمتر می‌شود.