به نام خدا

درس سیستم‌های عامل

نیم‌سال دوم ۰۱-۰۰

دانشکدهٔ مهندسی کامپیوتر

دانشگاه صنعتی شریف

مدرس **مهدی خرازی**

تمرین گروهی **یک ممیز صفر!**

موضوع **آشنایی با pintos**

موعد تحویل گزارش **ساعت ۲۳:۵۹ دوشنبه ۱۶ اسفند ۱۴۰۰**

با سپاس از دستیاران آموزشی: **علی احتشامی، مجید گروسی، عرشیا اخوان و امیرمهدی نامجو**

اقتباس شده از **CS162 در بهار ۲۰۲۰ در دانشگاه کالیفرنیا، برکلی**

فهرست مطالب

به اولین بخش از اولین تمرین از تمرین‌های گروهی درس خوش آمدید :)
در این درس از سیستم عامل آموزشی **pintos** استفاده می‌کنیم. دلیل استفاده از این سیستم عامل این است که با توسعه‌ی هسته‌ی یک سیستم عامل کوچک ولی واقعی، درکی عینی از مفاهیمی که داخل کلاس درس یاد می‌گیرید پیدا کنید. سیستم عامل **pintos**، نواقص زیادی در سامان دادن به پرونده‌ها [1]، زمان‌بندی بین ریسه‌ها [2] و اجرای برنامه‌های کاربر [3] دارد. شما در تمرین‌های گروهی این درس، این سیستم عامل را توسعه خواهید داد و ویژگی‌های مذکور را به آن اضافه خواهید کرد.

۱ راه‌اندازی

۱.۱ دریافت کد pintos

کد pintos در مخزنی با آدرس زیر قرار گرفته است:

```
https://tarasht.ce.sharif.ir/ce424-002-students/ce424-002-pintos
```

شما می‌توانید به طور جداگانه این مخزن را در یک پوشه ذخیره کنید و کدهای داخل آن را به صورت دستی به مخزن گروهتان منتقل کنید. همچنین می‌توانید مانند تمرین‌های فردی، آن را به صورت یک remote به نام handouts در پوشه‌ی مربوط به تمرین‌های گروهی‌تان اضافه کنید.
به عنوان راه دوم، می‌توانید دستور زیر را بررسی کنید و یا به فایل vm_patch.sh که در تمرین فردی ۰ در اختیارتان قرار گرفت مراجعه کنید.

```
git remote add
```

۲.۱ راه‌اندازی مخزن تمرین‌های گروهی

در ماشین مجازی خود به مسیر زیر بروید:

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

سپس فرمان زیر را اجرا کنید:

```
git clone https://tarasht.ce.sharif.ir/ce424-002-students/ce424-002-groupX .
```

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 آشنایی با pintos

برای این‌که بتوانید در تمرین‌های گروهی بعدی طراحی و پیاده‌سازی مناسبی انجام دهید نیاز به آشنایی با ساختار این سیستم عامل دارید. هدف این تمرین کمک برای آشنایی اولیه شما با کدهای **pintos** است.
هم‌چنین توصیه اکید می‌شود که همگام با انجام فعالیت‌های زیر به بخش چهارم مستند ذکر شده در قسمت منابع رجوع کرده و با روش‌های عیب‌زدایی در **pintos** آشنا شوید.

۱.۲ یافتن دستور معیوب

در ابتدا، دستورات **make** و **make check** را در پوشه‌ی **pintos/src/userprog** اجرا نمایید. طبیعتا می‌بینید که هم‌اکنون هیچ تستی به طور موفقیت‌آمیز اجرا نمی‌شود. ما در این تمرین قدم به قدم تست **do-nothing** را در **GDB** اجرا می‌کنیم تا در **pintos** تغییری ایجاد نماییم که در نتیجه‌ی این تغییر، این تست با موفقیت اجرا شود. همچنین، با روند اجرای برنامه‌های کاربر در این سیستم عامل که برای تمرین گروهی اول لازم است آشنا شوید.

[1] File System
[2] Thread Scheduling
[3] User Programs

تست **do-nothing** ساده‌ترین تست برای آزمودن توانایی **pintos** در پشتیبانی از برنامه‌های کاربر است. با خواندن کد **pintos/src/tests/userprog/do-nothing.c** می‌بینید که این کد، کد یک برنامه است که هیچ کاری انجام نمی‌دهد و تنها کد خروج ۱۶۲ را به سیستم عامل می‌دهد. ما عدد ۱۶۲ را به جای ۰ انتخاب کرده‌ایم تا دنبال کردن این مقدار در هنگام اجرای 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).

حال محتوای داخل **pintos/src/userprog/build/tests/userprog/do-nothing.result** را ببینید. این پرونده نشان‌دهنده‌ی خروجی تست **do-nothing** است که توسط چارچوب آزمون [4] **pintos** اجرا شده است. چارچوب آزمون 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 می‌شود، **pintos** این خروجی را نمی‌دهد و به جای آن، تست **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 فردی آشنا شدید) **disassemble** 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 [5] اجرا شده است، پس کد در **do-nothing.c** یا در یکی از دو پوشه‌ی **pintos/src/lib** و **pintos/src/lib/user** است.) در مورد هر دستوری که در قسمت قبلی **disassemble** 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 [6] ندهید، بلکه سعی کنید جوابی سطح بالاتر و مفهومی بدهید.

۲.۲ به سوی Crash

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

```
pintos-gdb ./kernel.o
```

سپس 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 می‌کنید پردازنده هنوز شروع به کار نکرده‌است. به صورت سطح‌بالا موارد ذیل قبل از این که **pintos** پردازه **do-nothing** را اجرا نماید اتفاق می‌افتند:

[4] Testing Framework
[5] User space
[6] Registers

- 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** [۷] تغییر می‌دهد و تابع **main()** را صدا می‌زند **(pintos/src/threads/init.c)** .
- تابع **main()** ، **pintos** را آماده به کار می‌کند. برای این کار **Scheduler** ، **Memory Subsystem** ، **Interrupt** ، **Vector** ، دستگاه‌های سخت‌افزاری [۸] و سامانه‌ی پرونده‌ها [۹] 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** را با فراخواندن

```
process_wait (process_execute ("do-nothing"));
```

از **run_task()** اجرا می‌کند. 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. مقادیر **eip** و **esp** را در ساختار **if_** تنظیم می‌کند. مقادیر ساختار **if_** را در مبنای 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؟

۱۲. بعد از اجرای **iret** ، دستور **info registers** را وارد نمایید تا محتویات ثبات‌ها 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 نماد[۱۰]های هسته را می‌خواند. حال که در **userspace** هستیم، باید نمادها را 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.

[7] https://en.wikipedia.org/wiki/Protected_mode
[8] Hardware Device
[9] File System
[10] Symbol

۳.۲ Debug

حال که دستور معیوب را پیدا کرده‌اید، هدف آن را فهمیده‌اید و قدم به قدم با روند اجرای برنامه توسط هسته آشنا شدید، 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 در واقع 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 **Interrupt Vector Frame** 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 قسمت 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 داده بودید، 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.

۳ گزارش نهایی

۱.۳ تحویل دادنی‌ها

شما باید به ۱۹ پرسش مطرح شده در قسمت قبل پاسخ دهید. برای این‌کار باید پرونده‌ی **markdown** قرار داده شده داخل مخزن تمرین گروهیتان را تکمیل کنید. هم‌چنین در صورت مشکل در نمایش کلمات انگلیسی و فارسی می‌توانید از این ابزار کمک بگیرید. هم‌چنین انتظار می‌رود با پاسخ به این پرسش‌ها در نهایت تست‌های **do-nothing** و **do-stack-align** را پاس کرده باشید.

۲.۳ نحوه نمره‌دهی

علاوه بر صحت گزارش نهایی، در طول تمام تمرین‌های گروهی موارد دیگری نیز بر نمره‌دهی گزارش شما موثر هستند: 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.
هم‌چنین 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)
- 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 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

اکیدا توصیه می‌شود برای آشنایی دقیق‌تر و عمیق‌تر با ساختار **pintos** و آشنایی با ساختار حافظه و تست‌های آن و هم‌چنین نحوه کار با ابزار دیباگ در آن، بخش ۴ (منابع) از این سند را مطالعه کنید.