Задание 1
Введение в xv6

Полезная литература
Более подробно разобраться с xv6 поможет книга R. Cox, F. Kaashoek, R. Morris «xv6: a simple, Unix-like teaching operating system» (https://pdos.csail.mit.edu/6.828/2021/xv6/book-riscv-rev2.pdf) (на английском).
Также, возможно, вам понадобится информация об архитектуре RISC-V, на которой запускается xv6.
Спецификация ISA (https://riscv.org/technical/specifications/) доступна на официальном сайте.

Часть 1. Pingpong
Прежде чем перейти к основной части курса, познакомимся подробнее с xv6 и её системными вызовами. Мы попробуем написать немного user-space кода.

Научитесь обмениваться данными между процессами с помощью специальных FIFO-каналов — Unix pipes (https://en.wikipedia.org/wiki/Pipeline_(Unix)).
Реализуйте программу `user/pingpong.c`, которая должна:

- Создать пайп.
- Создать дочерний процесс.
- Отправить сообщение `ping` из родительского процесса в дочерний.
- Прочитать их в дочернем процессе, вывести `<child pid>: got <message>` и отправить сообщение

pong в ответ.

- Прочитать ответ в родительском процессе, вывести `<parent pid>: got <message>`.

Советы для выполнения задания:

- Вам понадобится несколько системных вызовов — `pipe, fork, read, write, getpid`. Воспользуйтесь утилитой `man`, чтобы узнать, что делают эти вызовы и как ими пользоваться — поведение в xv6 не особо отличается от других Unix-подобных операционных систем.
- Вместо привычных вам `stdlib.h` и `stdio.h` доступна местная стандартная библиотека — `user/ulib.c` (user/ulib.c), а также `printf.c` (user/printf.c) и `umalloc.c` (user/umalloc.c). Посмотрите на другие программы в директории `user/` (user/), чтобы понять, как ей пользоваться.
- Добавьте программу в `UPROGS` в `Makefile`, чтобы она скомпилировалась.
- В программах для xv6 обязательно нужно вызывать `exit(0)` для выхода.

Часть 2. Dump

В прошлой части ЛР мы использовали системные вызовы, например, `pipe` и `fork`. Задача системных вызовов — дать программам из user-space возможность выполнять привилегированные команды.
Реализуем новый системный вызов `dump`. Он будет выводить на экран состояние регистров `s2–s12` вызывающего процесса.

Чтобы системный вызов был доступен из user-space, добавим в файл `user/user.h` (user/user.h) объявление функции `dump`, как это сделано для других системных вызовов. В файл `user/usys.pl` (user/usys.pl) добавьте строку `entry("dump")` — он отвечает за генерацию ассемблерных инструкций для совершения системного вызова.

Теперь реализуем сам системный вызов. Для этого добавьте функцию `dump` в файл `kernel/proc.c` (kernel/proc.c). Текущий процесс можно получить с помощью функции `myproc`. Структура proc содержит поле `trapframe`, в котором и находятся значения всех регистров. Избегайте лишней копипасты при выводе регистров.
Все регистры в xv6 64-битные, однако в рамках данного задания для каждого регистра вам нужно вывести лишь младшую 32-битную часть.

Наконец, отредактируйте файлы `kernel/syscall.h` (kernel/syscall.h), `kernel/sysproc.c` (kernel/sysproc.c) и `kernel/syscall.c` (kernel/syscall.c) так, чтобы появилась возможность вызвать `dump` из user-space. Посмотрите, как реализованы другие вызовы, и сделайте аналогично.

Осталось собрать xv6. Запустите утилиту `dumptests` (user/dumptests.c) и сравните фактические значения регистров и результат вашего системного вызова.

- Функцию dump нужно также определить заголовочном файле в `kernel/defs.h` (kernel/defs.h).
- Системный вызов должен возвращать 0 при успешном завершении, и код ошибки в остальных случаях. Наш системный вызов всегда завершается успешно.
- Если при запуске `dumptests` выводится сообщение о том, что системный вызов `dump` не найден, то попробуйте пересобрать xv6 с нуля.

Часть 3*. Dump2

Мы бы могли использовать системный вызов `dump`, чтобы написать собственный отладчик. Однако, у него есть два недостатка. Во-первых, он выводит значение

регистров на экран, и мы не можем обработать эти значения в user-space. Во-вторых, он позволяет узнать значения регистров только у текущего процесса, что делает невозможным отладку другого процесса. Напишем ещё один системный вызов, чтобы исправить эти недостатки — `dump2`.

У этого вызова будет три аргумента:

- `int pid` — номер процесса, для которого запрашивается значение регистра
- `int register_num` — номер регистра, число от 2 до 11
- `uint64 *return_value` — адрес, по которому необходимо вернуть значение

Обратите внимание, что в целях безопасности регистры процесса может смотреть только сам процесс и его родитель.

В этом системном вызове, в отличие от `dump`, вам понадобится корректно обрабатывать и возвращать ошибки:

- Верните `-1`, если вызвавший процесс не имеет прав смотреть требуемый регистр
- Верните `-2`, если процесса с таким идентификатором не существует
- Верните `-3`, если передан некорректный номер регистра
- Верните `-4`, если не удалось записать данные по переданному адресу

Полезные советы:

- Аргументы в системные вызовы передаются немного иначе, нежели в обычные функции. Посмотрите на другие системные вызовы, чтобы понять, как получать аргументы из user-space.
- Вы не можете записать данные по адресу `*return_value` — это виртуальный адрес в user-space, и использовать его в kernel-space невозможно. Вам поможет функция `copyout`.

Запустите `dump2tests` (user/dump2tests.c). Проверка происходит автоматически.

Требования к сдаче ЛР преподавателю:

- Наличие отчета, который включает в себя ссылку на репозиторий, вывод о проделанной работе
- Готовность запустить тесты по просьбе преподавателя