Задание 3. Copy-on-write fork

Системный вызов `fork` в текущей реализации копирует всю память родительского процесса в дочерний. Однако, как показывает практика, эта операция неэффективна: как правило, ни родительский, ни дочерний процесс не редактируют большую часть скопированной памяти, поэтому они могли бы пользоваться одной общей областью физической памяти.

Ещё хуже происходит в самом типичном сценарии использования `fork`:

```
int status = fork();
assert(status == 0);

const char *child_argv[] = {"/bin/some/program", "--help", NULL};
const char *child_envp[] = {NULL};
execve(child_argv[0], child_argv, child_envp);
```

Все страницы дочернего процесса незамедлительно уничтожаются. Кроме этого, сам процесс копирования занимает продолжительное время. Однако, совсем без копирования не обойтись — его можно делать в том случае, если один из процессов пытается что-то записать в область памяти. Такая оптимизация называется `copy-on-write`.

Наша цель — реализовать `copy-on-write` для системного вызова `fork`.

Часть 1. UB-on-write

Начнём с простого шага: уберём копирование памяти. Системный вызов `fork` (kernel-space код находится в kernel/proc.c#L244) для этого вызывает функцию `uvmcopy` (kernel/vm.c#L320). Давайте упростим её — просто запишем в новую таблицу страниц те же физические адреса, что и в
старую.

Как вы думаете, что пойдет не так? Попробуйте запустить обновленную версию и проверьте ваши предположения.

Часть 2. Профессиональные средства отладки

Напишите функцию `void vmprint(pagetable_t)` в kernel/vm.c, которая будет печатать содержимое таблицы страниц.

Вывод должен быть отформатирован следующим образом:

```
page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
.. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000
.. .. ..0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
.. .. ..1: pte 0x0000000021fda00f pa 0x0000000087f68000
.. .. ..2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
.. ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
.. .. ..510: pte 0x0000000021fdd807 pa 0x0000000087f76000
.. .. ..511: pte 0x000000002000200b pa 0x0000000080008000
```

Многоточиями (`..`) обозначаются уровни вложенности, в каждой строчке показывается индекс записи, PTE и физический адрес записи.

Чтобы не делать всё с нуля, воспользуйтесь макросами в конце файла kernel/riscv.h, посмотрите на функцию `freewalk` (kernel/vm.c#L284). Не забудьте добавить определение функции в kernel/defs.h, чтобы воспользоваться ей в нужных частях ядра.

Вызовите эту функцию в `exec.c` и выведите таблицу страницу для первого процесса. Обратите внимание: для всех остальных процессов выводить таблицу не нужно.

Часть 3. Fault-on-write

Для реализации оптимизации воспользуемся следующим трюком: запретим и родительскому, и дочернему процессу писать в продублированные страницы. Если они всё же попробуют это сделать, процессор сгенерирует отказ страницы, который

обрабатывается кодом ядра.

На этом шаге измените флаги PTE и запретите запись в страницы дочернего процесса, удалив флаг `PTE_W`. Перед запуском попробуйте представить, что произойдет. Запустится ли вот такая программа из шелла?

```
#include "user/user.h"

int main() {
    exit(0);
}
```

Попробуйте запустить ОС и `cowtest` в ней.

Часть 4. Copy-on-write

Переходим к самой важной части — копированию. Давайте теперь запретим запись как в страницы родительского, так и дочернего процесса в uvmcopy. Назовём такие страницы заблокированными.

Теперь при каждом обращении на запись в заблокированную страницу будет происходить отказ страницы. Мы можем обработать его в функции `usertrap` (kernel/trap.c#L36) — из всех ошибок нас интересует только отказ страницы — его код (`r_scause()`) равен 13 или 15. Обратите внимание на аргументы `printf` (kernel/trap.c#L71). Вам нужен регистр `stval` (`r_stval()`) — в нём лежит виртуальный адрес, по которому произошла ошибка.

Возьмите код из старой версии uvmcopy и таким же образом скопируйте заблокированную страницу, после чего отобразите старые виртуальные адреса в новые физические. Не забудьте разрешить запись в новую страницу.

- Используйте макрос `PGROUNDDOWN` — вам нужен не адрес с ошибкой, а адрес страницы.
- Если что-то падает, поищите значение `sepc` в kernel/kernel.asm — оно отвечает за адрес инструкции, на которой произошла ошибка. Не бойтесь, с ассемблером разбираться не придётся: вместе с инструкциями там есть строки вашего исходного кода.

Вам, возможно, понадобится информация о том, какие страницы заблокированы. Для этого можно использовать биты `RSW` в `RISC-V PTE` — восьмой и девятый.

Освобождайте физические страницы в kernel/kalloc.c только тогда, когда они никем не используются.

- Вам понадобится хранить счётчик ссылок. Делать это с помощью массива фиксированной длины (похожего на тот, от которого мы избавлялись в прошлом задании) — один из подходящих вариантов.

Помните, что если заблокированную страницу использует только один процесс, то её нужно просто разблокировать — так вы избежите утечки памяти.

Последний штрих — используйте ту же идею в copyout (kernel/vm.c#L365).

- В конце файла kernel/riscv.h вы найдете полезные макросы.

Убедитесь, что `cowtest` и `usertests` проходят. Первый тест из `cowtest`, к примеру, аллоцирует больше половины доступной памяти и делает `fork`.

- Вы можете убивать процесс, если при page fault на copy-on-write-странице в памяти не оказалось места.

Часть 5*. Ленивая аллокация

Программы хранят различные данные, как правило, в двух местах — в куче и на стеке. Чтобы получить место в куче, используется системный вызов `sbrk`. Он увеличивает место, выделенное для программы, на указанное количество байт.

Нередко программы аллоцируют память «на всякий случай», фактически её не используя. Некоторые операционые системы обрабатывают аллокации лениво — лишь запоминают новое адресное пространство, выделяя страницы только при необходимости. В качестве дополнительного задания нам необходимо добавить эту возможность в `Xv6`.

5.1. Уберите аллокацию

Измените `sbrk` (kernel/sysproc.c#L42) так, чтобы он не выделял память. Это не сложнее, чем подзадание 1.

Попробуйте угадать, что произойдет при запуске системы, а затем проверьте ваше предположение.

5.2. Верните аллокацию

В этом задании снова нужно изменять `usertrap` (kernel/trap.c#L36`).

В этот раз вам понадобится код из `uvmmalloc()` (kernel/vm.c#L241) — а именно вызов функций `kalloc()` и `mappages()`, которые вызывались из `sbrk()` через `growproc()`.

Не забудьте модифицировать `uvmunmap()` (kernel/vm.c#L181) и не освобождать ту память, которая не была фактически выделена.

Помимо этого, вам нужно будет обработать некоторое количество граничных случаев:

- Отрицательные аргументы `sbrk`
- Page-fault на неаллоцированной странице при нехватке памяти, а также на ещё не запрошенной странице (процесс тоже нужно убивать)
- Проверить, что копирование памяти в `fork()` работает корректно
- Корректно обрабатывать передачу ещё не выделенных адресов в системные вызовы — например, `read`. Всё это проверяется с помощью `lazytests` и `usertests`.
- Скорее всего, у вас не пройдут тесты `sbrkbasic` и `sbrkfail`. Посмотрите на функцию `wait` (kernel/proc.c#L384) и подумайте, как это исправить.

Отправка

В своем форке репозитория поместите изменения в ветку с названием в точности `lab-3` и сделайте MR в ветку `main`.