Задание 2. Аллокатор

В `Xv6` реализован только страничный аллокатор — выделять объекты размером меньше страницы неэффективно: на их хранение уйдет целая страница. Из-за этого, все «маленькие» объекты — файловые дескрипторы, структуры процессов и так далее — выделяются статически (kernel/file.c#L17):

```
struct {
    struct spinlock lock;
    struct file file[NFILE];
} ftable;
```

Таким образом, количество файлов ограничено переменной `NFILE`. Сильно увеличивать эту переменную нельзя — большая часть памяти уйдёт на работу операционной системы. Наоборот, если `NFILE` слишком мало, система не сможет поддерживать достаточное количество одновременно открытых файлов. (`NFILE`, как и многие другие константы, можно найти в файле kernel/params.h).

Наша цель — избавиться от этого, заменить аллокатор памяти и выделять файловые структуры динамически. Свой аллокатор писать не потребуется — buddy allocator уже написан. Для этого задания также заменен kernel/kalloc.c: там уже инициализируются служебные структуры аллокатора, где хранится информация о свободных и занятых блоках.

Часть 1. Использование аллокатора

Используйте для выделения файловых структур новый аллокатор вместо статического массива. Для этого измените файл kernel/file.c.

1. Теперь вам не нужен статический массив файловых структур — он находится в kernel/file.c#L19. При каждом вызове функции `filealloc` просто выделите место под файл с помощью `bd_malloc`.
2. Не забудьте освободить используемую память в функции `fileclose`.
   - Нужен ли теперь `ff`?
   - Нужна ли блокировка в `ftable.lock`? Зачем она вообще используется?
3. Чем заполнена область памяти, которую возвращает `bd_malloc`? Что было при использовании статического аллокатора?

После внесения правок в код запустите user/alloctest.c. Должен пройти первый тест:

```
$ alloctest
filetest: start
filetest: OK
...
```

Этот тест открывает больше, чем `NFILES` файлов и ожидает, что они все откроются.

Если вы просто увеличите `NFILE`, тесты не пройдут. Вы можете изменить kernel/file.c так, чтобы использовать не `NFILE`, а большую константу, но задание не будет принято, несмотря на пройденные тесты.

Запустите `usertests` и убедитесь, что все тесты проходят. Перед повторным запуском `usertests` вероятно вам понадобится вернуть файловую систему в изначальное состояние: `rm fs.img; make qemu`.

Часть 2. Оптимизация аллокатора

`Buddy allocator` можно оптимизировать следующим образом: сейчас мы для каждого блока храним два бита — флаги «блок занят» (`alloc`) и «блок поделен на две части» (`split`). Фактически, флаг «блок занят» мы используем только в одном месте — когда хотим понять, нужно ли объединить соседние блоки при освобождении.

Давайте попробуем здесь сэкономить 1 бит: вместо флага «блок занят» будем хранить для пары соседних блоков флаг «блок А занят хог блок В занят». Тогда флаг будет установлен, когда занят ровно один из двух блоков, и снят, если оба свободны или оба заняты. Когда мы занимаем или освобождаем блок, достаточно просто инвертировать флаг. При этом, если при освобождении блока бит изменился с 1 на 0, это означает, что до этого у нас из пары был занят ровно один блок, а теперь оба блока свободны. И именно в этом случае мы должны объединить блоки.

Если `Xv6` управляет объемом памяти размером 128 МБ, то такая оптимизация сохранит нам порядка 1 МБ памяти.

Примените эту оптимизацию для написанного аллокатора и проверьте, что свободной памяти становится больше. Для этого измените файл kernel/buddy.c.

1. Используйте `bd_print`, чтобы в любом месте увидеть состояние структур аллокатора. Обратите внимание, что если вы поменяете назначение массива `alloc`, то функция будет работать некорректно. Исправлять её необязательно, но это поможет вам отлаживать код.
2. Обратите внимание на то, как аллокатор инициализируется: он считает, что управляет объемом памяти, являющимся степенью двойки — немного большим, чем доступный объем памяти — однако, блок в начале (та часть, где хранятся служебные структуры) и блок в конце (который на самом деле недоступен) помечаются как выделенные.

После выполнения этой части снова запустите `alloctest`. Должны пройти оба теста.

```
$ alloctest
filetest: start
filetest: OK
memtest: start
memtest: OK
```

Второй тест проверяет, сколько памяти занимает ваш аллокатор: программа выделяет всю доступную память и проверяет, что вы сэкономили достаточное количество памяти.

Снова убедитесь, что все тесты из `usertests` проходят.

Часть 3*. Использование аллокатора

Примените `buddy allocator` для структур процессов, аналогично части 1. Тесты на эту часть отсутствуют, оно проверяется преподавателем вручную. Не забудьте проверить, что usertests работают.

Отправка

В своем форке репозитория поместите изменения в ветку с названием в точности `lab-2` и сделайте MR в ветку `main`.