

Московский Физико-Технический Институт (ГУ)

Диссертация допущена к защите
зав. кафедрой

_____ Тормасов Александр Геннадьевич

«_____» _____ 2019 г.

ДИССЕРТАЦИЯ
на соискание ученой степени
МАГИСТРА

**Тема: Перехват файловых операций для создания
виртуальных sparse-файлов с удаленным хранилищем в
операционной системе macOS**

Направление: 03.04.01 – Прикладные математика и физика

Магистерская программа: 111111 – Название программы

Выполнил студент гр. 376 _____ Копырин Денис Валерьевич

Научный руководитель,

д. ф.-м. н., ст. н. с.

_____ Костюшко Алексей

Консультант по ,

к. т. н., доц.

_____ Додь Алексей

Москва – 2019

Оглавление

Введение	4
Глава 1. Файловые операции	6
1.1. Файловая Система	6
1.2. Структура vnode	7
1.3. Операции над vnode	10
1.4. Операция mmap: файловый кэш	11
1.4.1. Unified Buffer Cache как часть vnode	12
1.4.2. Memory Objects	13
1.4.3. Paging info	14
Глава 2. Перехват VFS операций	18
2.1. Описание перехватов	18
2.2. Vnode перехваты	19
2.2.1. Способы перехвата	19
2.2.2. Поиск полей vnode	20
2.3. KAUTH авторизация	27
Глава 3. Представление виртуального файла	32
3.1. Карта регионов	32
3.2. Список непересекающихся регионов	34
3.2.1. Термины	35
3.2.2. Метод merge	36
3.2.3. Метод add	37
3.2.4. Метод checkAndRemove	39
3.3. Использование	44

Глава 4. Структура UserSpace клиента	46
4.1. Транспорт Kernel - User	46
4.1.1. IOUserClient	46
4.1.2. IOSharedDataQueue	48
4.1.3. Синхронный транспорт	50
4.2. Схема запросов/ответов	52
Глава 5. Имплементация и производительность	55
5.1. Связи объектов	55
5.2. Схема потока данных	56
5.3. Защита от фиктивного чтения	56
5.3.1. Списки фильтрации приложений	56
5.3.2. Проверка запущенных процессов	57
Заключение	60
Список литературы	61

Введение

В данной работе рассматривается способ работы с файлами, находящимися на удаленном хранилище, доступ к которым требуется осуществлять при помощи локальных средств работы с файлами (POSIX API). Подобная задача появляется при синхронизации пользовательских изменений с облаком, а также при работе с общими файлами многими пользователями.

Сформулируем задачу в виде требований к продукту.

1. Стриминг - возможность поблочного скачивания файла
2. Консистентность - после скачивания части файла, данные должны быть кэшированы локально

Классическое решение данной проблемы подразумевает использование виртуальной файловой системы таких как *smbfs* или *nfs* и их монтирование в папку - новый корень файловой системы. Такой подход удовлетворяет первому свойству, однако для обеспечения сохранения данных необходимо производить существенные изменения в протокол или добавлять новый уровень абстракции персистентного кэша.

В дипломе будет рассмотрено другое решение, не требующее создания искусственной файловой системы, для работы достаточно загрузки расширения ядра операционной системы macOS, *Kernel Extension (kext)*. В системе будут создаваться *обычные* sparse-файлы, не хранящие по умолчанию данных, а *kext* будет скачивать данные на операции чтения. Такой подход будет удовлетворять требованию 1 и 2.

Первая глава диплома описывает общую структуру работы файлов в ядре. В главах 2-4 приводятся разработанные методы и алгоритмы, которые необходимы для перехватов. В последней главе представлены схемы связей компонентов вместе и измерения характеристик продукта.

Глава 1

Файловые операции

1.1. Файловая Система

Виртуальная Файловая Система (Virtual File System/VFS) - подсистема, которая отвечает за работу с различными *Файловыми Системами (ФС/FS)* и находится между специфичной для FS и независимой от FS кодом, таким образом абстрагируя любые различия между внутренним устройством FS и остальной частью ядра. Ядро использует VFS для осуществления *ввода/вывода (I/O)* через *vnode*, которые являются обобщением файла в ядре и хранилищем метаданных.

Рассмотрим принципиальную схему работы VFS. В данной работе нас не интересует схема регистрации новых устройств и точек монтирования, но внутреннее устройство VFS по созданию и удалению новых *vnode*, а также передача данных от слоя к слою.

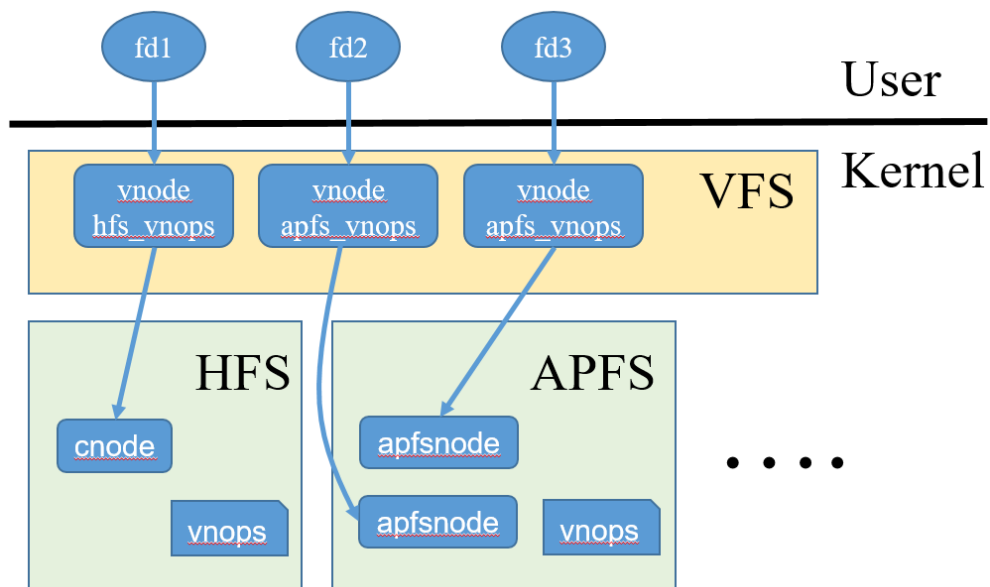


Рис. 1.1. Принципиальная схема работы VFS.

Рассмотрим 1.1. Каждая файловая система имеет в себе *единственный экземпляр* таблицы *vnops*, которая отвечает за работу с конкретной файловой системой. Приложение в UserSpace работает с файловыми дескрипторами (fd1-fd3 на рисунке), через которые производится доступ к vnode в VFS. Операции над файлами транслируется в один из вызовов VFS, который использует соответствующую функцию из таблицы vnops.

1.2. Структура vnode

vnode в системе может представлять из себя множество различных данных, как хранимых локально на диске, так и в памяти. Ограничим круг рассматриваемых vnode.

В таблице 1.2 перечислены возможные типы vnode в операционной системе macOS. Несмотря на некоторое сходство с FreeBSD, ядро XNU имеет некоторые отличия.

VREG; VDIR	Регулярный (обычный) файл; Директория
VBLK; VCHR	Блочное/Символьное устройство
VLNK	Символическая ссылка на файл
VSOCK; VFIFO	Сокет/Именованный pipe
VSTR; VCPLX	Специальные типы

Рис. 1.2. Типы vnode в VFS.

Для нашего рассмотрения ограничимся VREG и VDIR, однако можно включить в рассмотрение и VSTR, которые могут иметь под собой пользовательские данные.

Структура данных, описывающую vnode представлена в 1.1. Из определения удалены поля, которые не нужны для дальнейшего рассмотрения. Заметим, что данная структура также не является постоянной и меняется от

версии к версии в macOS. В данной работе будет использоваться ядро версии xnu-4903.221.2 [1], предложенные методы были протестированы на версии macOS до 10.10 включительно. В последующих главах будут предложены способы предсказания изменений структуры ядра, смотреть ??.

```
typedef struct vnode {
...
    int32_t v_kusecount;           /* count of in-kernel refs */
    int32_t v_usecount;           /* reference count of users */
    int32_t v_iocount;           /* iocounters */
...
    uint32_t v_id;                /* identity of vnode contents */
    union {
        struct mount    *vu_mountedhere; /* ptr to mounted vfs (VDIR) */
        struct ubc_info *vu_ubcinfo;     /* valid for (VREG) */
    } v_un;
...
    int (**v_op)(void *);        /* vnode operations vector */
    mount_t v_mount;             /* ptr to vfs we are in */
    void* v_data;                /* private data for fs */
} *vnode_t;
```

Листинг 1.1. Структура vnode в ядре, краткое содержание

У каждой *vnode* существуют два счетчика, которые отвечают за время жизни *vnode* в памяти. *usecount* - счетчик, обеспечивающий слабое наличие *vnode* в системе, *vnode* с положительным *usecount* не может быть переиспользована для другого файла. Слабое наличие файла не гарантирует наличие данных, если файловая система должна быть отмонтирована, например

vnode может быть прописано состояние VNON и данные *vnode* не могут быть больше использованы. *iocount* гарантирует, что *vnode* не только обязана соответствовать файлу, но и запрещено отмонтирование ФС пока *iocount* не будет равен 0. Отсюда требование VFS заключается в том, что *iocount* на *vnode* может быть взят только на короткое время операций, такие как чтение или запись блока данных. Kext обязан опускать *iocount* как только операции закончились, в противном случае система не может быть остановлена валидно. Для проверки переиспользования *vnode* используется $v_i d$, уникальный для данной *vnode* как указателя *id*, при изменении содержимого *vnode* счетчик поднимается.

v_mount и *v_data* отвечают за *mount*, к которому относится данная *vnode* и для расширения ядра являются непрозрачными (opaque) указателями. В данной работе особый интерес составляет поле *v_op*, которое отвечает за доступ к функциям работы с *vnode*, принадлежащей к ядру. Как уже было замечено раньше, указатель *v_op* является постоянным для всех *vnode* принадлежащих одной файловой системе. Для всех версий ядра от macOS 10.10 до macOS 10.14 сохраняется свойство последовательности полей $v_op \rightarrow v_mount \rightarrow v_data$, которое может быть использовано для нахождения оффсета *v_op* в памяти. Более подробно поле *v_op* изучено в главе 1.3.

В объединении *v_un* находятся указатели, являющиеся прозрачными для *vnode*. Содержимое указателя зависит от типа *vnode*, в данном дипломе будет рассмотрено содержимое указателя *vu_ubcinfo*, которое отвечает за кэширование данных *vnode*, смотреть 1.4.1.

1.3. Операции над vnode

Рассмотрим функция *VNOP_READ*, которая осуществляет чтение из файла. Остальные функции вида *VNOP_** работают аналогично. В листинге 1.2 представлена упрощенная функция [1].

Обратим внимание, что для доступа к операции в функции используется структура *vnop_read_desc*, ее сокращенное описание приведено в 1.3. Эта структура доступна публично в *KPI*.

```

errno_t VNOP_READ(...args...)
{
    int _err;
    struct vnop_read_args a;

    a.a_desc = &vnop_read_desc;
    a.a_vp = vp;
    ...other inits...

    _err = (*vp->v_op[vnop_read_desc.vdesc_offset])(&a);

    return (_err);
}

```

Листинг 1.2. Упрощенная функция *VNOP_READ*

```

struct vnodeop_desc {
    int vdesc_offset;          /* offset in vector */
    const char *vdesc_name; /* a readable name for debugging */
}

```

```

int vdesc_flags;          /* VDESC_* flags */

int *vdesc_vp_offsets;    /* list ended by VDESC_NO_OFFSET */
int vdesc_vpp_offset;     /* return vpp location */
...
};

```

Листинг 1.3. Структура *vnodeop_desc*

Видно, что используя соответствующую структуру вида *vnodeop_func_desc*, где *func* является именем необходимой структуры, можно найти смещение в таблице функций *v_op*. Таким образом, для того чтобы переопределить операцию над *vnode* необходимо подменить функцию в таблице *v_op* найденную через структуру *vnodeop_func_desc*.

1.4. Операция mmap: файловый кэш

В данной секции будет рассмотрена имплементация *memory – mapped* - доступ к файлу как к страницам в виртуальной памяти. У *vnode* вызываются методы *pagein* для первого чтения, результат сохраняется в кэше; далее используется скаченная страницы на каждого последующего побайтового чтениях. Таким образом осуществляется оптимизация доступа к дисковому хранилищу через страницы памяти, организация *readahead* и *writebehind* политик.

Для обеспечения такой подгрузки данных файла в страницы, *Mach* часть ядра *xnu* предоставляет подсистему кэширования данных файла *UBC* и подсистему виртуальной памяти *vm_map*. Это рассмотрение будет необходимо для осуществления фильтрации пустых страниц, см 5.3.

1.4.1. Unified Buffer Cache как часть vnode

При открытии *vnode* осуществляется аллокация структуры *vu_ubcinfo* (1.1). В 1.4 приведено описание данной полей [1].

```
typedef struct ubc_info {
    memory_object_t ui_pager;          /* pager */
    memory_object_control_t ui_control; /* VM control for the pager */

    vnode_t ui_vnode;      /* vnode for this ubc_info */
    kauth_cred_t ui_ucred; /* holds credentials for NFS paging */
    off_t ui_size;         /* file size for the vnode */
    uint32_t ui_flags;     /* flags */
    uint32_t cs_add_gen;    /* generation count when
                           csblob was validated */
    ...
} *ubc_info_t;
```

Листинг 1.4. Структура *ubc_info*

Часть *UBC*, отвечающая за работу с памятью, находится в Mach части ядра, а именно в полях *ui_pager* и *ui_control*. При помощи операций над файлом *pagein* и *pageout* осуществляется чтения и запись файла на диска с использованием указанных полей.

UBC имеет доступные методы в *KPI*, которые позволяют регулировать кэширование. Самый важным методом является *ubc_msync*, отправка всех "грязных" страниц на диск. Если кэш был умышленно заполнен неверными страницами, то при помощи *ubc_msync* можно сделать кэш снова валидным. Стоит заметить, что при таком подходе, количество операций чтения с диска становится порядка $O(n^2)$ в виду того, что каждое чтения *байта* порождает

чтение *всей страницы*. Отсюда при чтении страницы размером *pagesize* байт, с диска будет считано $pagesize^2$ байт.

1.4.2. Memory Objects

Реализация VFS в операционной системе macOS очень близка к FreeBSD, но существует множество изменений, связанных в первую очередь с наличием подсистемы Mach, отвечающей за работу с памятью.

Тип *memory_object_t* является *обобщенным (unified)* типом для любых обращений к памяти, в том числе к диску, далее *Memory Object*. В *KPI Memory Object* является прозрачным указателем на структуру *memory_object*.

Работа с *Memory Object* подразделена на 2 независимые компоненты, зависящие от источников данных, *memory_object* и *memory_object_control*. Поля *mo_ikot* оставлены в структуре для поддержки работы с *UserSpace* клиентами. В объекте *memory_object_control* поле *moc_object* отвечает за выделение памяти в ядре и не зависит от операций над *vnnode*. Объект *memory_object* является зависимым от источника данных, управление ими осуществляется при помощи аналогичных коллбэков.

```
typedef struct    memory_object {
    mo_ipc_object_bits_t    mo_ikot;
    const struct memory_object_pager_ops *mo_pager_ops;
    struct memory_object_control *mo_control;
} *memory_object_t;
```

```
typedef struct memory_object_control {
    mo_ipc_object_bits_t moc_ikot;
    struct vm_object *moc_object;
```

```
} *memory_object_control_t;
```

Листинг 1.5. Структура *memory_object*

Для данной работы, достаточно рассмотреть структуру, отвечающую за получения данных из источника *vnode* - *vnode_pager*. Заметим, что у структуры присутствует заголовок, отвечающий за общую для всех *Memory Object* и частный для самого *vnode_pager* указатель на *vnode*.

```
typedef struct vnode_pager {
    /* mandatory generic header */
    struct memory_object vn_pgr_hdr;

    /* pager-specific */
    unsigned int    ref_count;      /* reference count */
    struct vnode     *vnode_handle; /* vnode handle */
} *vnode_pager_t;
```

Листинг 1.6. Структура *vnode_pager*

TODO привести крутую картинку

1.4.3. Paging info

В этой секции будет рассмотрена часть реализации виртуальной памяти, относящейся к *mtar* файлам. Для работы с файлами в удаленном хранилище необходимо предоставлять некоторым процессам доступ к файлам, но не подгружать данные. В виду кэширования страниц в *vm_object* нельзя отдавать невалидные страницы (смотреть 1.4.1), поэтому было выбрано решение *подмены* *vm_object* файла на *vm_object* с пустыми страницами.

Приложения для доступа к этому кэшу используют виртуальную па-

мать, адреса в которой отображаются на *vm_object_t* через карту виртуальной памяти (*vm_map_t*). Схема карты показана на рисунке (1.3).

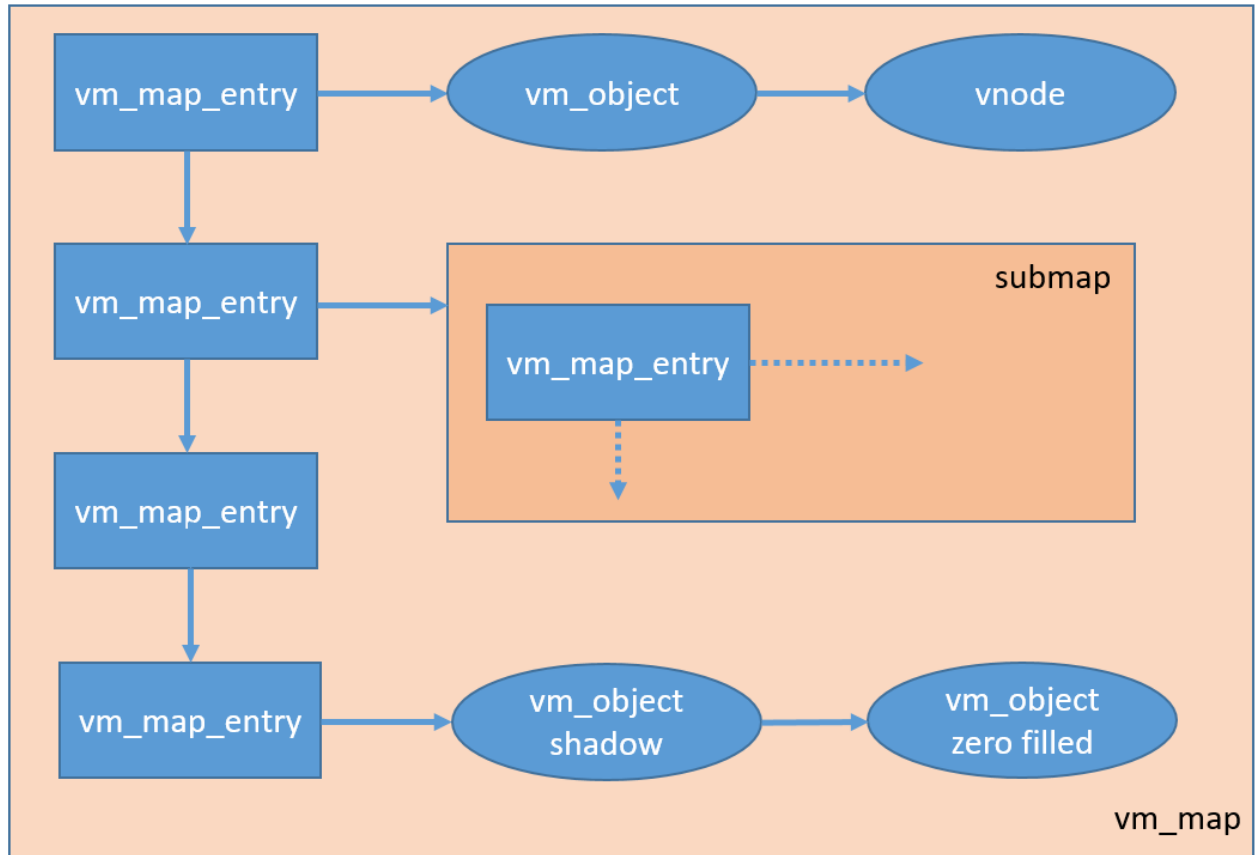


Рис. 1.3. Карта виртуальной памяти процесса

Карта состоит из списка записей. Запись определяет участок виртуальной памяти, к которому относятся адреса и указывает или на *vm_object* с страницами, или на другую *vm_map* - подкарта (*submap*). При использовании функции *fork* для обеспечения *copy-on-write* страниц виртуальной памяти используются *объекты с тенью*. Тень объекта - другой *vm_object*, который дает страницы для данного объекта при доступе.

Для получения карты *vm_map* используем функцию *current_map* и производим ее парсинг. В заголовке карты находится первый элемент списка записи. Для каждой записи ищем объект и проверяем принадлежит ли он к файлу с удаленным хранилищем. При нахождении объекта подменяем

указатель на новый *vm_object* с пустыми страницами. Структуры данных представлены в листинге 1.7.

```

struct _vm_map {
    lck_rw_t lock;
    struct vm_map_header hdr;
    ...
}

struct vm_map_header {
    struct vm_map_links links; /* first, last, min, max */
    int    nentries; /* Number of entries */
    ...
};

struct vm_map_links {
    struct vm_map_entry *prev; /* previous entry */
    struct vm_map_entry *next; /* next entry */
    vm_map_offset_t    start; /* start address */
    vm_map_offset_t    end;   /* end address */
};

struct vm_map_entry {
    struct vm_map_links links; /* links to other entries */

    struct vm_map_store store;

    union vm_map_object vme_object; /* object I point to */
}

```

```

    vm_object_offset_t  vme_offset; /* offset into object */
    ...
}

typedef union vm_map_object {
    vm_object_t    vmo_object; /* object object */
    vm_map_t       vmo_submap; /* belongs to another map */
} vm_map_object_t;

```

Листинг 1.7. Структуры данных *vm_map*

Глава 2

Перехват VFS операций

2.1. Описание перехватов

Перехват/Перенаправление (Hook, Interception) - способ вставки собственных обработчиков до и после вызова функции. Псевдокод представлен на рисунке 2.1. Заметим, что в данном примере функция *BEFORE* может осуществлять фильтрацию вызова оригинальной функции при возврате ненулевого значения. Такой подход может понадобиться при блокировании доступа к файлу и возвращения *EPERM* на открытии.

```
int HOOK_FUNC(params)
{
    int ret = 0;
    ret = BEFORE(params);
    if (!ret)
        int ret = ORIGINAL(params);

    AFTER(params, ret);
    return ret;
}
```

Листинг 2.1. Псевдокод перехвата

2.2. Vnode перехваты

2.2.1. Способы перехвата

Для создания файлов с виртуальным хранилищем нужны перехваты многих *VNOP* функций у *vnode*. Рассмотрим возможные варианты вставки *hook* функции в *VNOP* функцию и их сравнение в таблице 2.1.

Сравнение	<i>v_op</i> in <i>vnode</i>	<i>func</i> in <i>mount v_op</i>
Перезаписи	$O(vnode)$	$O(mount * func)$
Аллокации	$O(mount * v_op)$	$O(1)$
Защита	отсутствует	Read Only
Производительность	перехват нужных <i>vnode</i>	перехват всех <i>vnode</i>

Рис. 2.1. Таблица способов перехвата

В таблице приведено 2 вида перехватов: перезапись всей таблицы *v_op* в нужной *vnode* и перезапись необходимой функции *VNOP* в таблице, хранящейся в *mount*.

Перезапись всей таблицы требует создание *полной копии v_op* и последующей вставки нового указателя в *каждую нужную vnode*. В копии *v_op* производится подмена указателей необходимых функций на *hook*. В альтернативном варианте *hook* вставляется непосредственно в таблицу, что не требует аллокации данных, но таблица в *mount* является защищенной с правами *ReadOnly* как статические данные в *kext*, отвечающий за работу с *FS*.

При перезаписи *v_op* только в нужных *vnode* можно обеспечить лучшую гранулярность и производительность. В случае перезаписи всей функции в таблице, необходимо производить фильтрацию операций для *vnode*, наблюдение за которыми не нужно. Подобную фильтрацию возможно осуществить при помощи *hashtable* в *kext* или *per vnode хранилища*.

Прямая запись в таблицу требует аккуратной работы при отгрузке *kext*. Драйвер, который имеет в себе необходимую таблицу может быть отгружен и указатели в его статические данные будет некорректным. В первом варианте подобная проблема не может возникнуть, так как память таблицы принадлежит *kext*, который осуществляет подмену указателей и при отгрузке достаточно пройти по всем *vnode* и произвести обратную запись оригинального указателя. Однако, при подмене *v_op* в *vnode* нужно следить за временем жизни *vnode*, при переиспользовании нужно восстановить оригинальный указатель на таблицу.

Последнее замечания накладывает существенные ограничения на второй вариант *hook*, поэтому в этой работе будет использована перезапись *v_op* в *vnode*, но с помощью дополнительных проверок можно использовать и второй вариант.

2.2.2. Поиск полей *vnode*

Эвристический метод

Поля структуры *struct vnode* не являются открытыми в *KPI*, так как поля *vnode* могут изменяться при смене версии ядра, но расширения ядра *XNU* должны работать на любой версии системы. Поэтому в *KPI* существуют обертки над полями структуры, такие как *vnode_parent* для получения *v_parent*, *vnode_tag* для получения *v_tag*, *vnode_setparent* для записи в *v_parent* и т.п. Не ко всем полям структуры есть доступ через *KPI*, в частности невозможно получить доступ к полю *v_op*. В этой секции будут рассмотрены способы нахождения оффсета *v_op* в структуре *struct vnode*.

Рассмотрим сначала простой вариант. В приватной структуре *struct vnode* (1.1) можно увидеть, что поля *v_op* и *v_mount* находятся на расстоянии 8 байт. Поле *v_mount* можно получить через функцию в *KPI* *vnode_mount*.

Производим поиск значения, которое возвращается в функции *vnode_mount* и находим оффсет в структуре. Далее вычитаем из сдвига 8 и получаем итоговый оффсет поля *v_op*. Пример функции, делающий поиск приведен в 2.2.

Заметим, что такое расположение полей в структуре *vnode* не является гарантированным и может измениться в следующих версиях системы.

```
// Requires fixed struct vnode

...

int (**v_op)(void*);

mount_t v_mount;

...

int find_vop_offset(vnode_t vp)
{
    void* needle = vnode_mount(vp);
    void** haystack = (void**) vp;
    int i;
    for (i = 0; ; i++)
    {
        if (haystack[i] == needle)
            break;
    }

    return (i - 1) * sizeof(void*);
}

VOPFUNC* get_vop(vnode_t vp)
```

```

{
    return (VOPFUNC*)((char*)vp + find_vop_offset(vp));
}

```

Листинг 2.2. Нахождения оффсета поля *v_op*

Динамическое дизассемблирование

Более надежный способ нахождения оффсетов полей в структуре - использование динамического дизассемблера в ядре, например distorm3. Для этого рассматривается скомпилированный код функции, осуществляющей доступ к нужному полю через операции разыменования, такие как *mov...* и *lea*. Будем такой парсинг функции называть *эмуляцией*.

Рассмотрим скомпилированный код функции *VNOP_CREATE* в различных вариантах ядра *xnu*. В *release* 10.14.2 версии *amd64* код ассемблера приведен в листинге 2.3.

```

mov     rbp, rsp
push    rbp
...
push    rbx
sub     rsp, 48h
lea     rax, __stack_chk_guard
mov     rax, [rax]
mov     [rbp+__dtrace_args], rax
mov     r15, r8
mov     r12, rdx
mov     rbx, rsi
mov     r13, rdi

```

```

lea    rax, vnop_create_desc kernels
mov     [rbp+var_70], rax
mov     [rbp+var_68], r13
mov     [rbp+var_60], rbx
mov     [rbp+var_58], r12
mov     [rbp+var_50], rcx
mov     [rbp+var_48], r15
mov     rax, [r13+0xD0]
movsxd  rcx, cs:vnop_create_desc.vdesc_offset
lea     rdi, [rbp+var_7]
call    qword ptr [rax+rcx*8]

```

Листинг 2.3. Дизассемблированный код *VNOP_CREATE*, *release* версия ядра

Функций *VNOP_CREATE* получает информацию о таблице *v_or* через первый аргумент функции *dup*. В соглашении о вызовах *System V AMD64 ABI* принято передавать первый аргумент через параметр *rdi*. Будем следить за регистрами в которые будет перемещено значение *rdi*. Легко увидеть, что такими регистрами являются только *r13* и *rdi*. Далее для получения значения *v_or* используется инструкция `mov rax, [r13+0xD0]` и искомое значения смещения является `0xD0`. Отсюда требуется найти инструкцию вида `mov REG0, [REG1+OFFSET]`, где *REG1* - регистр, за которым производится наблюдение. Далее по коду можно увидеть вызов функции *create* из таблицы `call qword ptr [rax+rcx*8]`, что является подтверждением корректности.

Теперь рассмотрим ту же функцию *VNOP_CREATE* в ядре *debug 10.13.6*. *Debug* версии *xnu* имеют существенное большее число проверок и являются менее оптимизированными, но подобный код упрощает отладку ядра с символами. Листинг приведен в 2.4.

```
push    rbp
mov     rbp, rsp
sub     rsp, 80h
lea     rax, [rbp+var_70]
lea     r9, _vnop_create_desc
lea     r10, ___stack_chk_guard
mov     r10, [r10]
mov     [rbp+var_8], r10
mov     [rbp+var_18], rdi
mov     [rbp+var_20], rsi
mov     [rbp+var_28], rdx
mov     [rbp+var_30], rcx
mov     [rbp+var_38], r8
mov     [rbp+var_70], r9
mov     rcx, [rbp+var_18]
mov     [rbp+var_68], rcx
mov     rcx, [rbp+var_20]
mov     [rbp+var_60], rcx
mov     rcx, [rbp+var_28]
mov     [rbp+var_58], rcx
mov     rcx, [rbp+var_30]
mov     [rbp+var_50], rcx
mov     rcx, [rbp+var_38]
mov     [rbp+var_48], rcx
mov     rcx, [rbp+var_18]
mov     rcx, [rcx+0D0h]
movsxd  rdx, dword ptr [r9]
```

```

mov    rcx, [rcx+rdx*8]
call   rcx

```

Листинг 2.4. Дизассемблированный код *VNOP_CREATE*, *debug* версия ядра

Для сохранения отладочной информации данные сохраняются не только в регистрах, но и на стеке, поэтому необходимо следить за перемещениями переменной *rdi* на стеке тоже. Будем обозначать переменные на стеке как $rdp + OFFSET$. Легко увидеть, что используемые переменные будут *rdi*, $rbp + 0x18$, *rcx* (временно), $rbp + 0x68$, *rcx*. Искомая инструкция аналогична *release* случаю. Также заметим, что подтверждающая инструкция требует поддержки загрузки эффективного указателя *lea*, которая используется для загрузки *vnop_create_desc*.

Опишем методы использования дизассемблера *distorm3* для эмуляции функций. Будем называть *уровнем* количество разыменований переменной. Считаем, что поле без разыменований находится на нулевом уровне.

Рассмотрим двумерный массив, индексируемый по регистрам/оффсетам стека и уровню, хранящий оффсет на текущем уровне и оффсет предыдущего уровня. Изначально заполним всю таблицу значением -1, что будет соответствовать неиспользованной переменной.

Ассемблерные инструкции производят сдвиги полей в таблице аналогично динамическому программированию. Любые операции, имеющие в качестве цели регистр выставляют -1 в оффсет данного уровня, так как значение регистра перезаписано. Далее для каждого уровня производим сдвиг, зависящий от инструкции, смотреть таблицы 2.1, 2.2, 2.3. Эмуляция останавливается по условию, например вызов функции с параметрами. Пользователь считывает поля таблицы для требуемого регистра и получает оффсет в структуре.

В примерах приняты обозначение оффсетов $\frac{A}{B}$, *A* - текущий уровень, *B* - предыдущий уровень.

MOV RBX, RDI

deref	<i>rax</i>	<i>rbx</i>	<i>rcx</i>	<i>rdx</i>	<i>rdi</i>	<i>rsi</i>	...
0		$\frac{18}{8}$ ←			$\frac{18}{8}$...
1		$\frac{28}{0}$ ←			$\frac{28}{0}$...
2		$\frac{-1}{-1}$			$\frac{-1}{-1}$...
3		$\frac{-1}{38}$			$\frac{-1}{-1}$...

Таблица 2.1. Сдвиги полей при перемещении

LEA RBX, [RSI + o]

deref	<i>rax</i>	<i>rbx</i>	<i>rcx</i>	<i>rdx</i>	<i>rdi</i>	<i>rsi</i>	...
0		$\frac{8 + o}{8}$ ←			$\frac{8}{8}$...
1		$\frac{-1}{0}$			$\frac{-1}{18}$...
2		$\frac{-1}{-1}$			$\frac{-1}{-1}$...
3		$\frac{10 + o}{0}$ ←			$\frac{10}{0}$...

Таблица 2.2. Сдвиги полей при перемещении с оффсетом

MOV RAX, [RDI + o]

deref	<i>rax</i>	<i>rbx</i>	<i>rcx</i>	<i>rdx</i>	<i>rdi</i>	<i>rsi</i>	...
0	$\frac{-1}{8}$				$\frac{-1}{8}$...
1	$\frac{-1}{0}$				$\frac{0}{18}$...
2	$\frac{0}{0 + o}$ ←				$\frac{10}{0}$...
3	$\frac{0}{10 + o}$ ←				$\frac{-1}{-1}$...

Таблица 2.3. Сдвиги полей при разыменовании

Подобный способ является более надежным, чем предположение о фиксированности поля *v_mount*, однако требует эмуляции функции и рассчитывает на ограниченный круг используемых функций *mov* и *lea* в данном случае. Расширение поддерживаемых инструкций эмуляции, например обнуление регистра при помощи *xor*, может дать более точные результаты.

2.3. KAUTH авторизация

KAUTH авторизация позволяет *kext* ограничивать доступ в определенным *vnode* в системе, осуществлять мониторинг операций, происходящих в системе и, существенно для данной работы, получать первичные сведения о *vnode* в системе. [2]

KAUTH предоставляет слушателям (*listener*) области (*scope*) наблюдения, в которых функции ядра отправляют запросы на авторизацию события. Клиент для принятия решения получает *vnode*, полномочия (*credentials*) и действие (*action*). *Credentials* позволяют получить информацию о *UserSpace* приложении, производящем действие. Функция авторизации *listener* возвращает одно из трех решений, перечисленных в таблице 2.2, префикс опущен.

DEFER	Решение не принято, передать следующему <i>listener</i>
ALLOW	Разрешить операцию, других <i>listener</i> не вызывать
DENY	Запретить операцию, других <i>listener</i> не вызывать

Рис. 2.2. Коды возврата *listener*

Функции регистрации на событие и обработчика событий *listener* описаны в 2.5.

```
// Register for events from scope
```

```

kauth_listener_t kauth_listen_scope(
    const char* identifier,
    kauth_scope_callback_t callback,
    void* idata);

// Unregister listener from events
void kauth_unlisten_scope(
    kauth_listener_t listener);

typedef int (*kauth_scope_callback_t)(
    kauth_scope_t scope,
    kauth_cred_t credential,
    kauth_action_t action,
    uintptr_t arg0,
    uintptr_t arg1,
    uintptr_t arg2,
    uintptr_t arg3
);

```

Листинг 2.5. Функции регистрации *KAUTH*

Ядро *XNU* предоставляет два стандартных *scope* для работы с файлами: *FileOp scope* и *Vnode scope*. *FileOp* формально не является областью наблюдения для авторизации, но только предоставляет сведения об операциях, происходящих в системе. *Vnode scope* дает контроль над авторизацией и позволяет запрещать операции, но слушатели *Vnode scope* вызывается на порядок чаще *FileOp scope*.

В таблицах 2.3 и 2.4 представлены все , которые необходимо обработать функции авторизации. Операции из *Vnode scope* могут быть совмещены вме-

сте в битовой маске *action*. Снова обратим внимание, что *FileOp scope* вызовы осуществляются *после* события, поэтому возврат *listener* функции будет проигнорирован. Однако во время *FileOp* вызовов все данные уже проинициализированы, поэтому можно осуществить in-memory сканирование.

OPEN	Файл $(vnode_t)arg0$ был открыт по пути $(char*)arg1$
CLOSE	Файл $(vnode_t)arg0$ по пути $(char*)arg1$ был закрыт
RENAME	Файл по пути $(char*)arg0$ переименован в $(char*)arg1$
EXCHANGE	Был произведен обмен данных файлов (exchangedata) по путям $(char*)arg0$, $(char*)arg1$
LINK	Была создана жесткая ссылка (<i>Hard Link</i>) к файлу по пути $(char*)arg0$ по пути $(char*)arg1$
EXEC	Запущена программа с $vnode_t(arg0)$ по пути $(char*)arg1$

Рис. 2.3. *FileOp* действия

При работе с *Kauth* необходимо помнить о фильтрации собственных запросов, в противном случае произойдет deadlock при попытке авторизовать действие под замком. Простейшее решение проблемы является фильтрация путей *vnode*. Подобное решения является допустимым, но имеет ряд очевидных недостатков. Во-первых, пути фильтрации могут пересекаться с другими файлами, например при фильтрации доступа к файлу */var/db/data* необходимо отсекаать аутентификацию папки */var/db*, что недопустимо в общем случае. Во-вторых, тривиальное решение имеет плохую расширяемость, если пути прописаны в исходном коде.

Универсальное решение данной проблемы может быть использование новых *credential*, созданных *kext* из существующих *kauth_cred* и *vfs_context*: будем называть такие контексты *чистыми* (*Clean*). Получить *vfs_context* можно из функции *vfs_kernel_context* и *kauth_cred* из *vfs_context_ucred*

В таблице используются обозначения

$vp = (vnode_t)arg1$, $dvp = (vnode_t)arg2$.

$arg0$ - *VFS context*, *credential* при авторизации.

$arg3$ - код ошибки при *KAUTH_RESULT_DENY*.

READ_DATA	Будет прочитано содержимое файла vp
WRITE_DATA	Будет записано содержимое файла vp
EXECUTE	Будет запущен файл vp
DELETE	Файл vp в папке dvp будет удален
READ_ATTRIBUTES	Будет прочитаны атрибуты файла vp
WRITE_ATTRIBUTES	Будет записаны атрибуты файла vp
READ_EXTATTRIBUTES	Будет прочитаны расширенные атрибуты (xattr) файла vp
WRITE_EXTATTRIBUTES	Будет записаны расширенные атрибуты (xattr) файла vp
READ_SECURITY	Будет прочитаны Access Control List (ACL) файла vp
WRITE_SECURITY	Будет записаны Access Control List (ACL) файла vp
TAKE_OWNERSHIP	Будет изменен владелец файла vp
LINKTARGET	Будет создан <i>Hard Link</i> файла vp
ACCESS	Специальный флаг для запроса возможности действия над файлом

Рис. 2.4. *Vnode* действия для *регулярных* файлов

для *vfs_kernel_context*. Доступ через полученные *credentials* нужно фильтровать для защиты от deadlock. Пример фильтрации представлен в листинге 2.6. Заметим, что подобный подход может быть применен только для запросов к локальному файловому хранилищу: для удаленных дисков(smbfs например)

root может иметь недостаточно прав для записи, в таком случае необходимо создавать множество чистых контекстов для разных точек монтирования.

```
typedef vfs_context {
    thread_t thread;
    kauth_cred_t cred;
} vfs_context_t;

void Prepare(vfs_context_t parent) {
    vfs_context_t CleanCtx = vfs_context_create(parent);
    kauth_cred_t CleanCred = kauth_cred_create(vfs_context_ucred(
        parent->cred));
    CleanCtx->cred = CleanCred;
}

bool ShouldSkip(vfs_context_t ctx, kauth_cred_t cred) {
    return ctx == CleanCtx || cred == CleanCred;
}
```

Листинг 2.6. Фильтрация Kauth запросов

Глава 3

Представление виртуального файла

3.1. Карта регионов

Для представления полученных данных на диске необходимо хранить карту блоков, которые можно отдать на чтения. В данном разделе будет рассмотрен структура данных, позволяющая работать с такой картой.

Рассмотрим действия над двумя картами A и B . *Объединение* ($A \cup B$) двух карт есть новая карта, которая содержит блоки обеих карт. *Разность* ($A \setminus B$) двух карт будем называть функцию, которая возвращает карту, которая содержит все блоки карты A за исключением блоков карты B . Действия *пересечение* является действие вида $A \setminus (A \setminus B)$.

Для имплементация класса *BlockRegions* введем рассматривать действия, которые будут изменять сам объект, но не возвращать новый экземпляр. Интерфейс представлен в листинге 3.1.

```
struct BlockRegion
{
    UInt64 start;
    UInt64 length;
};

class BlockRegionsIterator
{
public:
    virtual ~BlockRegionsIterator() {};
```

```

    virtual BlockRegion* getNext() = 0;
    virtual void reset() = 0;
};

class BlockRegions
{
public:
    virtual ~BlockRegions() {};

    virtual int add(BlockRegion) = 0;
    virtual bool checkAndRemove(BlockRegion) = 0;

    virtual BlockRegionsIterator* getIterator() const = 0;

    virtual bool isEmpty() const = 0;
    virtual void clear() = 0;
    virtual int getCount();
};

```

Листинг 3.1. Интерфейс класса *BlockRegions*

Блоки представлены в виде полуотрезков вида $[start, end)$, $start < end$, $length = start - end$, указывающие на часть файла, которая была скачена. Такой полуотрезок будем называть *регион* (*BlockRegion*).

BlockRegions дает интерфейс для работы с картой блоков регионов. Действия *объединение* и *разность* представлены в виде функций *add* и *checkAndRemove*. *checkAndRemove* также возвращает информация о существовании пересечения между регионом и картой.

BlockRegionsIterator предоставляет интерфейс для получения списка регионов, хранящихся в *BlockRegions*. *Iterator* можно использовать для удобного расширения методов *add* и *checkAndRemove* до ...(*BlockRegions**), смотреть листинг 3.2.

```
void BlockRegions::FUNC(BlockRegions* regions)
{
    BlockRegionsIterator* it = regions->getIterator();
    BlockRegion* region;
    while ((region = it->getNext()))
        FUNC(*region);

    delete it;
}
```

Листинг 3.2. Удобное расширение FUNC до списка регионов

3.2. Список непересекающихся регионов

BlockRegions никак не ограничивает тип хранимых блоков. Для данной работы необходимо, чтобы блоки не пересекались, в противном случае нельзя понять каких блоков не хватает для чтения из файла. Рассмотрим реализацию интерфейса *BlockRegions_Disjoint*, который хранит упорядоченные непересекающиеся регионы. Будем представлять набор регионов в виде односвязного списка, отсортированного по возрастанию *start*. Список организован через дополнительное хранимое поле в структуре (3.3). В классе добавим *sentinel* поле *Head*, которое будет отвечать за голову списка и будет пустым *BlockRegion_DisjointElem*. Новый метод *merge* будет "скле-

ивать "пересекающиеся элементы списка и будет показан позже.

```
class BlockRegions_Disjoint : public BlockRegions
{
    ...
protected:
    BlockRegion_DisjointElem* Head;
    virtual int merge(BlockRegion_DisjointElem* merge);
};

struct BlockRegion_DisjointElem
{
    UInt64 start;
    UInt64 length;
    BlockRegion_DisjointElem* next;
};
```

Листинг 3.3. Расширение структуры *BlockRegion*

3.2.1. Термины

Введем отношение порядка на регионах *BlockRegion_DisjointElem* по полю *start*: элемент *a* меньше элемента *b*, если $a.start < b.start$, аналогично для меньше или равно. *Sentinel* регион меньше любого другого региона.

Назовем состояние объекта *BlockRegions_Disjoint* валидным, если выполнены 3 условия:

1. Нет пересечений регионов,
2. Регионы в списке отсортированы по возрастанию,

3. Отсутствуют регионы нулевой длины.

Тогда можно представить наборы блоков валидного объекта в виде отрезков на оси (рис. 3.3).

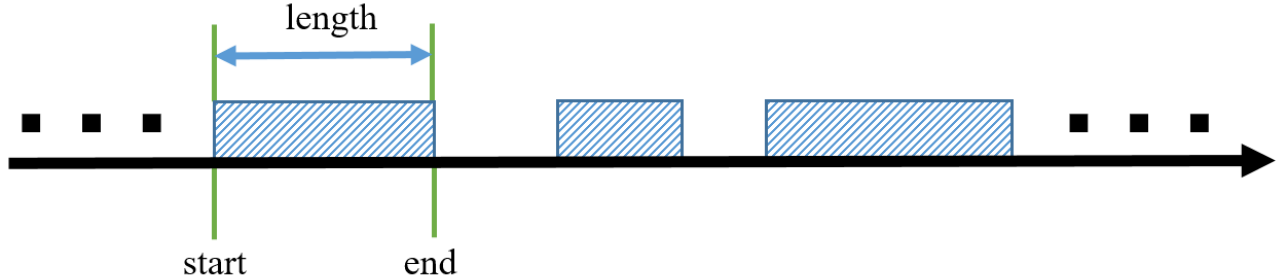


Рис. 3.1. Обозначения для регионов на оси

Будем считать, что объект *BlockRegions_Disjoint* обязан быть в валидном состоянии после выполнения любого метода класса.

В листингах для удобства будут использованы макросы *START* и *END*, определенные в 3.4.

```
#define START(reg) ((reg)->start)
#define END(reg) ((reg)->stat + (reg)->length)
```

Листинг 3.4. Макросы доступа в *BlockRegion*

3.2.2. Метод `merge`

Метод *merge* является защищенным методом и помогает восстановить список из невалидного состояния при нарушении свойства 1: наличие пересекающихся регионов, при этом будем требовать выполнения свойств 2 и 3. Аргумент функции *merge* - регион *victim*, после которого происходит нарушение свойства 1, то есть регионы *victim* и *victim* \rightarrow *next* пересекаются. Псевдокод метода *merge* приведен в листинге 3.5.

```

void merge(BlockRegion* victim)
{
    if (victim == sentinel) return;

    // Find if next is intersecting
    while (victim->next != NULL && END(victim) >= START(victim->next))
    {
        // Merge the region and adjust victim
        merged = region->next;
        if (END(victim) < END(victim->next))
            victim->length = END(victim->next) - START(victim);

        victim->next = merged->next;
        delete merged;
    }
}

```

Листинг 3.5. Псевдокод функции *merge*

Метод *merge* проверяет, что следующий регион $victim \rightarrow next$ пересекается с *next* и изменяет размер *victim*, если требуется. Так как выполнено регионы отсортированы (свойство 2), то *while loop* достаточно для проверки всех необходимых регионов. Заметим, что не всегда необходимо изменять длину региона *victim*: $victim \rightarrow next$ находится полностью внутри *victim* не требует изменения длины.

3.2.3. Метод *add*

Пусть метод *add* при вставке региона *new* будет находить такой наибольший элемент в списке, что его значение *start* меньше или равно *start*

вставляемого региона. Обозначим найденный регион *last*. Заметим, что в виду существования *sentinel* региона, *last* всегда будет найден. Тогда возможно два варианта расположения нового региона, показанных на рисунке (3.2): $end(last) > start(new)$ или $end(last) \leq start(new)$. Эти два случая соответствуют наличию или отсутствию пересечения *last* и *new*.

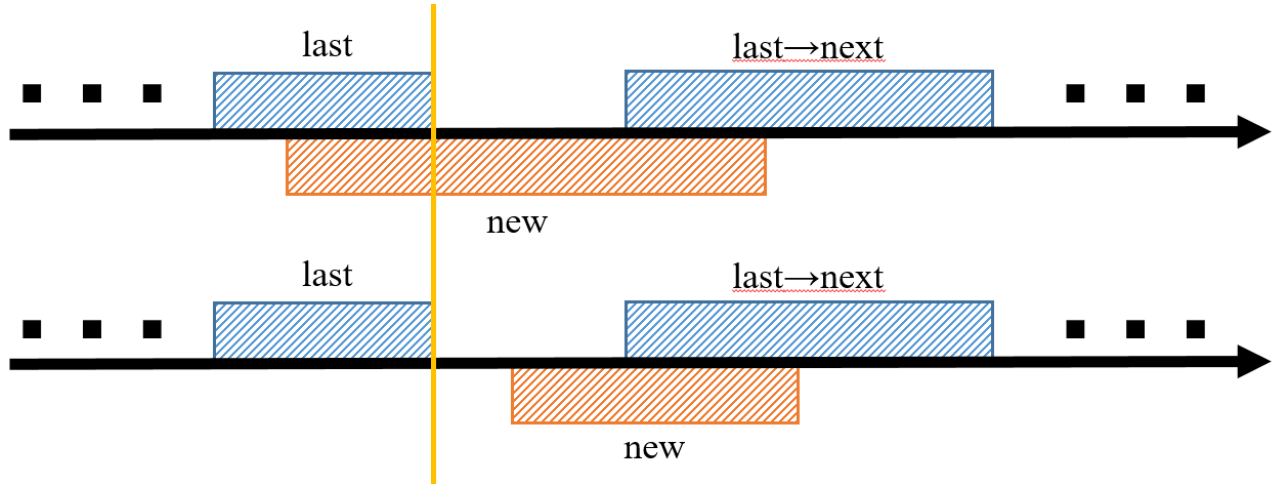


Рис. 3.2. Варианты для добавления нового региона

Из рисунка можно увидеть, что данное состояние списка является невалидным, так как свойство 1 валидности не выполнено. При вставке после элемента *last* другие свойства остаются выполненными, поэтому достаточно функцию *merge* с правильным регионом, *next* которого пересекается с собой: *last* в первом случае, *new* во втором случае.

Так получаем итоговый псевдокод функции *add* (листинг 3.6). Дополнительная проверка в начале была добавлена для гарантии свойства отсутствия нулевых регионов.

```
void add(BlockRegion new)
    if (new->length == 0)
        return;
```

```

// Find last region
BlockRegion_DisjointElem* last = Head;
while (last->next != NULL && START(last->next) < START(new))
    last = last->next;

// Insert in list and do merge
new->next = last->next;
last->next = new;

if (START(new) <= END(last))
    merge(last);
else
    merge(new);
}

```

Листинг 3.6. Псевдокод функции *add*

3.2.4. Метод `checkAndRemove`

Для имплементации *checkAndRemove(remove)* будем идти по списку регионов и сравнивать его с *remove* регионом. Рассмотрим всевозможные способы пересечения двух регионов. Будем считать, что регион над осью - регион из списка, регион под осью - *remove* регион.

Заметим, что при удалении из валидного списка не может нарушиться свойства 1 и 2 валидности, так как новые регионы не создаются и сортировка оставшихся регионов не нарушена.

Рассмотрим варианты пересечения регионов и распишем как создать новую карту. Будем использовать обозначение *region* для текущего региона из списка.

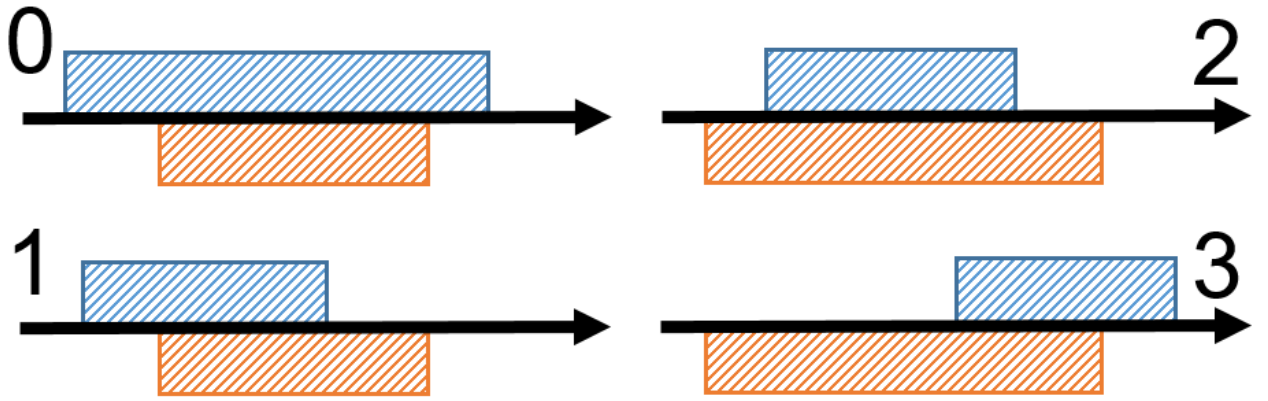


Рис. 3.3. Способы пересечения двух регионов

0. $start(region) \leq start(remove) \ \& \ end(remove) \leq end(region)$.

Регион *remove* лежит внутри *region*. Итоговый список теперь должен иметь 2 региона: до *remove* и после *remove*. Заметим, что данная операция требует создания нового региона, а регион *region* можно переиспользовать для обозначения региона до *remove*. Подобная процедура переиспользования будет применена в следующих случаях аналогично.

1. $start(region) < start(remove) < end(region) < end(remove)$.

Конец *region* находится в *remove*. В этом случае достаточно лишь поменять длину региона *region*. Так как условия строгие, то региона нулевой длины получиться не может.

2. $start(remove) \leq start(region) \ \& \ end(region) \leq end(remove)$.

Регион *region* полностью находится внутри региона *remove*. Достаточно удалить такой регион из списка. Обратим внимание, что данный случай покрывает краевые случаи равенства одного или двух концов.

3. $start(remove) < start(region) < end(remove) < end(region)$.

Начало *region* находится в *remove*. В этом случае меняем начало региона *region* на значения конца региона *remove*. Так как условия строгие,

то региона нулевой длины получиться не может.

Понятно, что данная процедура является корректной: всевозможные случаи расположения двух регионов, включая краевые случаи, рассмотрены. Также предположим, что случай равенства и начала, и конца регионов *region* и *remove* есть случай 3. Тогда, никакой регион не может удовлетворять одновременно двум случаям: если один из концов равен другому, то это может быть только вариант 0 для другого конца снаружи или вариант 2 для другого конца внутри. Если равенство отсутствует, то из-за строгих сравнений может быть выполнен лишь один случай. При этом легко убедиться в том, что если регионы не пересекаются, то есть $end(remove) \leq start(region)$ или $end(region) \leq start(remove)$, они не удовлетворяют ни одному из представленных условий.

Приведем псевдокод для данных условий в листинге 3.7. В цикле проходятся все регионы до конца *remove* и для каждого из них проверяются условия. Обратим внимание, на способ получения следующего региона: сохраняется значение *previousRegion*, через *next* которого получаем следующий в списке регион. Во время проверки условий *region* может быть удален, тогда выставляется новое значение *next* для *previousRegion*. Также заметим, что важно выставлять корректно новые *start* и *length* для переиспользованных регионов, *length* должен быть записан раньше, чем *start* из-за метода определения макросов *START* и *END*.

```
bool checkAndRemove(BlockRegion remove)
{
    // Do not try to remove empty region
    if (remove.length == 0)
        return false;
```



```
bool isInRegions = false;
```

```
BlockRegion region = Head->next;
```

```
BlockRegion previousRegion = Head;
```

```
// Optimization for definite not intersecting region
```

```
while (region != NULL && START(region) < END(remove)) {
```

```
    // Condition 0: Split existing region
```

```
    if (START(region) <= START(remove) && END(remove) <= END(region))
    {
```

```
        // Leftover region from right
```

```
        if (END(region) - END(remove) != 0) {
```

```
            BlockRegion leftoverRegion = new BlockRegion;
```

```
            leftoverRegion->start = END(remove);
```

```
            leftoverRegion->length = END(region) - END(remove);
```

```
            leftoverRegion->next = region->next;
```

```
            region->next = leftoverRegion;
```

```
        }
```

```
        // Leftover region from left is reused from region
```

```
        region->length = START(remove) - START(region);
```

```
        if (region->length == 0) {
```

```
            // This is set to make "region = previousRegion->next" command
            be universal
```

```
            previousRegion->next = region->next;
```

```
            delete region;
```

```

    region = previousRegion;
}

isInRegions = true;
}

// Condition 1: End is in check region -> move end
if (START(region) < START(remove) && START(remove) < END(region)
    && END(region) < END(remove)) {
    region->length = START(remove) - START(region);
    isInRegions = true;
}

// Condition 2: Current region is INSIDE check region -> delete,
// fix current region
if (START(remove) <= START(region) && END(region) <= END(remove))
{
    // Should be done before delete
    previousRegion->next = region->next;
    delete region;

    region = previousRegion;
    isInRegions = true;
}

// Condition 3: Current region is intersecting with check region
-> move start

```

```

if (START(remove) < START(region) && START(region) < END(remove)
    && END(remove) < END(region)) {
    // Order is important!
    region->length = END(region) - END(remove);
    region->start = END(remove);

    isInRegions = true;
}

previousRegion = region;
region = previousRegion->next;
}

return isInRegions;
}

```

Листинг 3.7. Псевдокод функции *checkAndRemove*

3.3. Использование

Список регионов *BlockRegion_Disjoint* можно хранить для каждого виртуального файла в *расширенном атрибуте (xattr)*. При попытке чтения из файла производится чтение данных из регионов, которые до этого были еще не считаны. Псевдокод получения несчитанных регионов приведен в листинге 3.8. Если количество отсутствующих регионов равно нулю, то запрос клиенту можно не производить и так оптимизировать работу.

```
BlockRegions getUnreadRegions(vnode_t vp, BlockRegion regToRead)
```

```
{  
    BlockRegions presentedRegions = REGIONS(vp);  
    BlockRegions unreadRegions;  
    unreadRegions.add(regToRead);  
    foreach (Region reg in presentedRegions)  
    {  
        unreadRegions.checkAndRemove(reg);  
    }  
  
    return unreadRegions;  
}
```

Листинг 3.8. Псевдокод функции *checkAndRemove*

Глава 4

Структура UserSpace клиента

4.1. Транспорт Kernel - User

Для работы виртуальных файлов нужен UserSpace клиент, который будет предоставлять содержимое удаленных документов. В этой главе будет рассмотрен транспорт, который позволит получать данные, само устройство клиента может быть любым и зависит от применения виртуальных файлов, это может быть облако или архив. [3]

Kext для пользователей в *UserSpace* является сервисом, предоставляющим информацию. Время жизни клиента должно определять поведение *kext*, например смерть клиента не должна приводить к ошибкам в ядре.

4.1.1. IOUserClient

При регистрации создается объект наследника класса *IOUserClient*. В листинге 4.1 приведен пример кода, который позволяет найти в дереве *IOKit kext*. Объект *io_connect* может быть использован для организации транспорта.

```
io_connect_t open_connection(char* driverName)
{
    kern_return_t ret = KERN_SUCCESS;
    CFDictionaryRef classToMatch = NULL;
    io_service_t serviceObject;
    io_connect_t connection;
    kern_return_t kr;
```

```

classToMatch = IOServiceMatching(driverName);
serviceObject = IOServiceGetMatchingService(
    kIOMasterPortDefault, classToMatch);
IOServiceOpen(serviceObject, mach_task_self(), type, &connection);
IOObjectRelease(serviceObject);
return connection;
}

```

Листинг 4.1. Нахождения *kext* в *IOKit* дереве

Для вызова методов ядра используется функция *IOConnectCallMethod*, которая копирует данные клиента и вызывает функцию *externalMethod* в *kext*. Клиент может передать и получить 2 типа данных, числа *uint64* или структуру, они упаковываются в объект *IOExternalMethodArguments*. *Kext* переопределяет функцию *IOUserClient :: externalMethod* и передает управление в зависимости от селектора (*selector*), заменяя статический экземпляр объекта *IOExternalMethodDispatch*. *IOUserClient :: externalMethod* проверяет валидность размеров переданных клиентом данных и вызывает функцию, указанную в *IOExternalMethodDispatch*. Заметим, что допустима передача данных в структуре переменной длины, для этого необходимо указать размер -1 в поле *check...*. В листинге 4.2 приведены прототипы функций в *Kernel* и *UserSpace*.

```

kern_return_t IOConnectCallMethod(mach_port_t connection, uint32_t
    selector,
    const uint64_t* input, uint32_t inputCnt,
    const void* inputStruct, size_t inputStructCnt,
    uint64_t* output, uint32_t* outputCnt,

```

```

void* outputStruct, size_t* outputStructCnt);

IOReturn IOUserClient::externalMethod(
    uint32_t selector, IOExternalMethodArguments* args,
    IOExternalMethodDispatch* dispatch,
    OSObject* target, void* reference);

struct IOExternalMethodDispatch
{
    IOExternalMethodAction function;
    uint32_t checkScalarInputCount;
    uint32_t checkStructureInputSize;
    uint32_t checkScalarOutputCount;
    uint32_t checkStructureOutputSize;
};

typedef IOReturn (*IOExternalMethodAction)(
    OSObject* target, void* reference,
    IOExternalMethodArguments * arguments);

```

Листинг 4.2. Прототипы структур и функций транспорта *kext* \rightarrow *user*

4.1.2. IOSharedDataQueue

При существующей связи *user* \rightarrow *kext* организация обратной связи *kext* \rightarrow *user* возможно многими способами. В данной работе будет рассмотрена очередь сообщения *IOSharedDataQueue* как транспорт *kext* \rightarrow *user*.

Kext создает экземпляр очереди с указанием размер, например, во время присоединении клиента. Клиент после инициализации создает порт для пе-

передачи данных через *IODataQueueAllocateNotificationPort*, вызывает функцию *IOConnectMapMemory* и получает указатель на разделенную память *IODataQueueMemory* для работы с очередью. Далее клиент ждет событий о добавлении новых данных при помощи *DataAvailable* и получает их при поступлении через *Dequeue*. Пример кода клиента представлен в листинге 4.3.

```
void ListenToEvents(io_connection_t connection)
{
    mach_port_t port = IODataQueueAllocateNotificationPort();
    IOConnectSetNotificationPort(connection, 0, port, 0);

    mach_vm_address_t address = 0;
    mach_vm_size_t size = 0;
    IOConnectMapMemory(connection, 0, mach_task_self(), &address, &
        size, kIOMapAnywhere);

    IODataQueueMemory *queueMemory = (IODataQueueMemory *)address;
    do {
        while (IODataQueueDataAvailable(queueMemory)) {
            void* msg = malloc(entry->size);
            kr = IODataQueueDequeue(queueMemory, msg, &entry->size);
            { /* Parse message here */ }
            free(msg);
        }
    }
    while (IODataQueueWaitForAvailableData(queueMemory, port) ==
```



```

kIOReturnSuccess);

IOConnectUnmapMemory(connection, 0, mach_task_self(), address);
}

```

Листинг 4.3. *UserSpace* код для работы с *IOSharedDataQueue*

Со стороны *Kernel kext* должен предоставить зарегистрировать порт, переопределяя метод *IOUserClient :: registerNotificationPort* и устанавливая его в созданную очередь при помощи метода *IOSharedDataQueue :: setNotificationPort*. Для отправки сообщений *kext* должен использовать метод *IOSharedDataQueue :: enqueue*, который *асинхронно* добавляет новый элемент в очередь.

4.1.3. Синхронный транспорт

Для работы виртуальных файлов на операции *Read* должна происходить *синхронная* загрузка данных, в противном случае *Read* считает невалидное содержимое файла. Сформулируем требования для транспорта, достаточные для организации виртуальных файлов.

1. Синхронность. *Kext* на операции чтения должен ожидать пока сервис не загрузит корректные данные файл,
2. Наличие тайм-аута (*timeout*). Иметь *timeout* удобно для отладки и уменьшения нагрузки на приложение, предоставляющее данные (режим *Passthrough*),
3. Очистка аллоцированных объектов. Все объекты должны быть освобождены после использования.

Ожидаемые прототипы функций представлены в листинге 4.4.

```

/// Sync header
struct SyncMsg {
    uintptr_t ptrSelf;
    uintptr_t time;
};

/// Sends msg to userspace and waits for UnlockSyncMsg
/// ptr & time in SyncMsg should not be filled
bool SendSyncMsg(struct SyncMsg* msg, UInt32 msgSize);

// Callback to unlock waiting sync msg
// Ptr should be the one filled in struct SyncMsg
bool UnlockSyncMsg(uintptr_t ptrSelf, uintptr_t time);

```

Листинг 4.4. Прототипы функций синхронного транспорта

Заметим, что очередь является асинхронной: метод *enqueue* не ждет пока клиент вызовет функцию *Dequeue*. Рассмотрим способ сделать синхронный транспорт из асинхронной очереди. Будем при отправке создавать *conditional variable* и *mutex*, которые будем разблокировать при получении нотификации об обработке. Функция *msleep* позволяет ожидать событие на *conditional variable*, заданной как произвольный указатель (*msleep channel*). Для простоты достаточно использовать сам указатель на аллоцированный *mutex* или указатель на сообщение, которое нужно отправить.

Для требования очистки будет предполагать, что функция, создавшая *mutex* должна его и очистить. В таком предположении получаем схему работы синхронного интерфейса с тайм-аутами, представленную на рисунке 4.1.

Нормальный режим работы представлен в первой временной секции: *kext* инициализирует сообщение обфусцированным указателем на *SyncMsg*, отправляет сообщение в очередь, *msleep* ожидает событие от *UnlockMsg*, клиент обрабатывает сообщение и вызывает *UnlockMsg*, *kext* очищает аллоцированные данные. Множественные *msleep* нужны для обнаружения смерти клиента, если никто не сможет вызвать *UnlockMsg*, то ожидание можно остановить.

При первом тайм-ауте, после $MsleepCount * Repeat$ секунд, берется последний *msleep* на время *Unlock Timeout* секунд. В это время у клиента есть последний шанс разблокировать *kext*. После *Unlock Timeout* секунд клиент больше не должен доступаться к аллоцированной памяти. *kext* при этом сам разблокирует *mutex* и ждет еще *Mutex Free Timeout*. Этот тайм-аут нужен для случая, когда *UnlockMsg* уже начал доступ к функции, а *Unlock Timeout* уже прошел. После окончания *Mutex Free Timeout* данные очищаются.

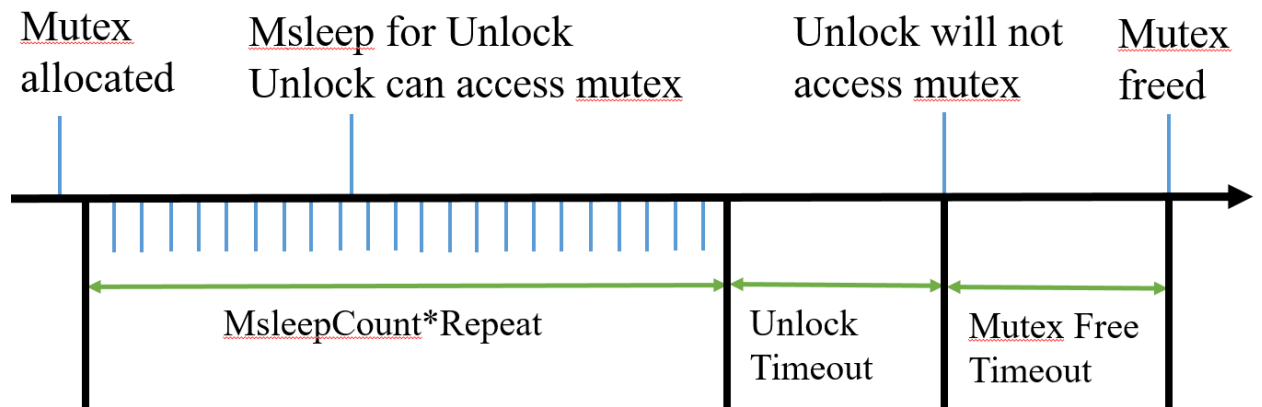


Рис. 4.1. Схема работы синхронного интерфейса

4.2. Схема запросов/ответов

Рассмотрим схему запросов и ответов для клиента и *kext*. Рабочий режим начинается при установке путей за которым необходимо следить. *Kext*

перехватывает операции записи и при вызове *VNOP_WRITE* отправляет сообщение клиенту с регионами, которые нужно скачать. Будем считать, что количество регионов ограничено восемью, при необходимости сообщение будет отправлено несколько раз. Также заметим, что для открытия файла достаточно знать его *vnode_id* и *dev_id*, однако путь является более переносимым. Структура сообщения представлена в листинге 4.5.

```
struct DownloadRequestMsg
{
    struct SyncMsg header;
#ifdef COMPAT
    const char path[MAXPATHLEN];
#else
    uint64_t vnodeId;
    uint64_t devId;
#endif
    BlockRegion[MAX_COUNT];
};
```

Листинг 4.5. Синхронное сообщение для загрузки регионов в файл

Клиент в ответ вызывает метод *kext* для разблокирования синхронного сообщения, *kext* разрешает чтения файла.

Опишем дополнительные методы управления *kext*. Для сохранения информации о регионах файла добавим функции для доступа в расширенным атрибутам или к *per vnode* хранилищу. *Kext* также будет кэшировать информацию о статусе файла, обеспечим доступ к кэшу через функции *addToCache* и *removeFromCache*. Для изменения содержимого файла со стороны клиента дадим функции блокировки операций чтения и записи *lockFileRead* и

lockFileWrite. Для очистки кэша после дегидратации, введем метод для удаления *UBC* кэша *invalidateVnodeCache*.

Глава 5

Имплементация и производительность

5.1. Связи объектов

В данной работе *userspace* процесс не будет рассматриваться конкретно в виду его сложности с точки зрения работы с данными. *Next* будет состоять из нескольких классов и их отношениях. Схема связей классов представлена на рисунке 5.1.

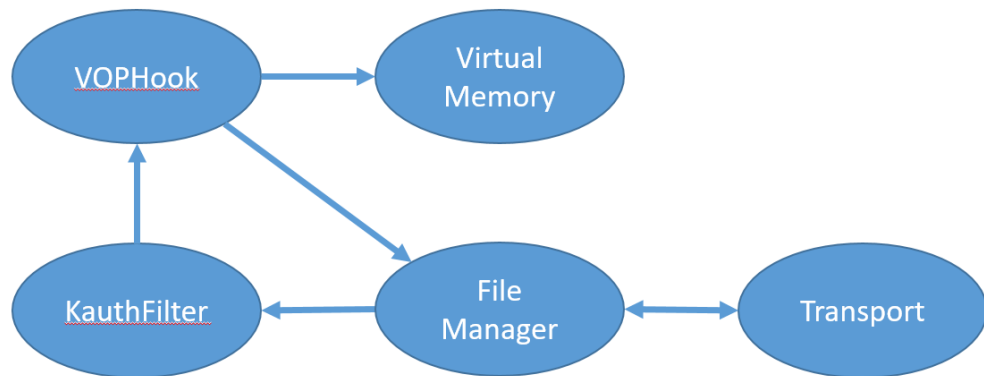


Рис. 5.1. Схема классов

При авторизации чтения из файла на *VnodeOp* коллбэке в *KauthFilter* необходимого файла производится проверка является ли файл виртуальным. Если файл виртуальный и его таблица *VOP* не перехвачена, производится ее перехват. На операции чтения из файла, в *File Manager* отправляется регион чтения. При помощи методов *Block Region* находится та часть файла, которую необходимо скачать с удаленного хранилища. Если пересечение не пусто, отправляется синхронный запрос *UserSpace* клиенту и ожидается пока в файл не будут записаны данные. На операции *pagein* в случае необходимости производится итерация по списку записей карты виртуальной памяти и подменяется *vm_object*.

Исходный код решения представлен в приложении 1 дипломной работы.

5.2. Схема потока данных

Структура сообщений, отправляемая пользовательскому приложению представлена в листинге 4.5. Для уменьшения количества отправляемых сообщений будем округлять размеры чтения регионов чтения. Рассмотрим два случая:

1. Побайтовое последовательное чтение,
2. Побайтовое чтение больше размера блока.

Будем считать, что количество считанных байт равно $Count$, размер блока равен $Block$. Количество сообщений, отправленных в *userspace* равно примерно $Count/Block$. Количество отправленных данных равно $Count/Block * sizeof(DownloadRequestMsg)$.

5.3. Защита от фиктивного чтения

Наибольшая проблема наличия виртуальных файлов в системе - их *фиктивное чтение*. Процессы, которые производят индексацию локальных файлов рассчитывают, что содержимое любых файлов является постоянным и чтение безопасно. Для виртуальных файлов подобное чтение является недопустимым, так как оно порождает полную выгрузку файла с удаленного хранилища.

5.3.1. Списки фильтрации приложений

Для разрешения данной проблемы будем производить изменение поведения чтения по *спискам фильтрации* двух категорий:

1. Запрет чтения файла процессам,
2. Отдача "пустых" данных.

При полном запрете чтения данных приложение никогда не будет получать некорректные данные, однако подобное поведение может породить некорректную работу. Так, при работе с *файлом, отображенным в память (memory-mapped file)* при запрете чтения приложение войдет в *deadlock* состояние на чтении байта или приведет к его падению.

При отдаче пустых данных файловая система может закэшировать фиктивные страницы и в последствии не вызвать функции чтения данных для других процессов. *UBC* при получении пустых страниц на операции *pagein* произведет их кэширование и тем самым будет требоваться последующая очистка кэша при помощи функции *ubc_msync*.

В операционной системе macOS приложения, отвечающие за кэширование являются *mds* и *mdworker*. Их можно безопасно включить во список фильтрации 1. *Finder* и *quicklookd* производят отрисовку иконок и предварительного просмотра изображений и видеоматериалов. Заметим, что *Finder* не даст пользователю открыть файл, если он сам не может его открыть с правами пользователя, поэтому *Finder* должен находиться во втором списке фильтрации.

5.3.2. Проверка запущенных процессов

Рассмотрим часть структуре процесса *struct proc*.

```
struct proc {
    ...
    char p_comm[MAXCOMLEN+1]; /* Process Command Name */
}
```



```

...
struct vnode *p_textvp;    /* Vnode of executable. */
off_t p_textoff;          /* offset in executable vnode */
...
}

```

Листинг 5.1. Структура процесса

Начальную фильтрацию можно производить при помощи *имени* процесса *p_comm*, но в системе может быть запускаемый файл с таким же именем, что будет порождать *false positive*. В структуре процесса также записана *vnode*, которая имеет в себе данные о запущенном приложении. Фильтрация по имени является достаточным признаком процесса, но является дорогостоящей операцией. Для уменьшения нагрузки можно использовать свойство уникальности файла в системе по 64-битному полю *vnode id* и 32-битному *device id*, вместе образующие 96-битный уникальный ключ *file id*.

Для нахождения *file id* в ядре достаточно открыть файл с необходимым именем, например */usr/bin/mdworker*, и получить его атрибуты, смотреть (5.2).

```

struct FileId
{
    uint32_t devId;
    uint64_t vnodeId;
};

FileId getFileId(vnode_t vp, vfs_context_t ctx)
{
    struct vnode_attr va;

```

```
vnode_getwithref(vp);  
  
VATTR_INIT(&va);  
VATTR_WANTED(&va, va_fileid);  
VATTR_WANTED(&va, va_fsid);  
  
vnode_getattr(vp, &va, ctx);  
vnode_put(vp);  
  
return { va.va_fsid, va.va_fileid };  
}
```

Листинг 5.2. Структура процесса

Заключение

В результате исследования было написано расширение ядра macOS, осуществляющее создание виртуальных файлов с удаленным хранилищем. При помощи перехватов операций *vnode* остается поддержка *POSIX API*, а файлы при скачивании остаются локально. Парсинг виртуальной памяти отфильтрованных процессов позволяет тонко контролировать доступ к невалидным страницам файла. Описанный синхронный протокол дает возможность гибкого соединения с *UserSpace* клиентом, при сбоях возможно использование тайм-аута и более простой проверки работоспособности.

Предложенные методы используются в Acronis для продуктов Acronis File Protect и Acronis Sync & Share для macOS.

Список литературы

1. Apple. macos xnu source code. — Access mode: <https://github.com/apple/darwin-xnu>.
2. Apple. Technical note tn2127: Kernel authorization. — Access mode: https://developer.apple.com/library/archive/technotes/tn2127/_index.html.
3. Singh A. Mac OS X Internals: A Systems Approach. — Addison Wesley Professional, June 19, 2006.