

# Рязанова, лекция 5.

Очевидно, что если постоянно обращаться к диску, то это очень большие временные затраты, что приводит к торможению приложений. Разработчики системы учитывают эти моменты - кэширование.

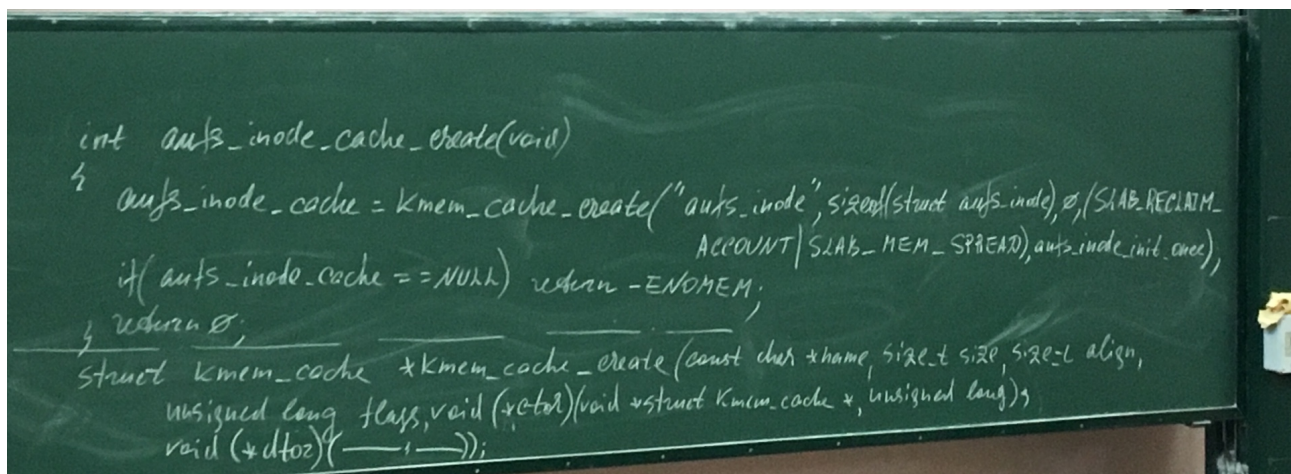
В силу того, что SLAB кэширование построено на принципе хранения данных одного размера, доступ к этим данным ускоряется.

В случае распределения SLAB участки памяти, подходящие для размещения объектов данных определенного типа и размера определены заранее. Распределитель (allocator) SLAB хранит информацию о размещении этих участков, которые известны также как кэши. Очевидно, что за счет того, что выделяются блоки памяти одного размера, ускоряется выделение памяти. SLAB аллокатор запрашивает у ОС большие участки памяти и выделяет из них небольшие участки по запросу. Вследствие этого обращение к менеджеру памяти выполняется реже, а запросы пользователя удовлетворяются быстрее. Но самым главным выигрышем является не скорость выделения памяти при использовании SLAB'ов, а сокращение времени на инициализацию такой памяти. Это связано с тем, что аллокатор часто используется для выделения объектов одного и того же типа, что позволяет пропустить инициализацию некоторых полей структур при повторном выделении такого участка памяти. Например, мьютексы, спинлоки, inode'ы и т. п. при освобождении объекта, скорее всего, имеют правильное значение. В силу этого при повторном выделении не нужны повторные инициализации части полей.

В настоящее время LINUX имеет три вида SLAB аллокаторов: SLAB, SLUB, SLOB. У них есть различия, но интерфейс у них один и тот же.

Рассмотрим некоторые функции SLAB аллокатора:

Эти функции применены для inode'ов



```
int aux_inode_cache_create(void)
{
    aux_inode_cache = kmem_cache_create("aux_inode", sizeof(struct aux_inode), 0, (SLAB_RECLAIM_
        ACCOUNT | SLAB_MEM_SPREAD), aux_inode_init_func);
    if (aux_inode_cache == NULL) return -ENOMEM;
    return 0;
}

struct kmem_cache *kmem_cache_create(const char *name, size_t size, size_t align,
    unsigned long flags, void (*ctor)(void *struct kmem_cache *, unsigned long),
    void (*dtor)(void *));
```

Сб, 13 апреля

const char \* name - имя кэша которое используется виртуальной файловой системой proc, которые можно посмотреть в /proc/slabinfo

sizeof(struct aufs\_inode) - размер объекта

size\_t align - выравнивание

Это callback функции, которые пишутся разработчиком

Функция kmem\_cache\_create не выделяет память кэшу

```
void aufs_inode_cache_destroy(void)
{
    rcu_barrier();
    kmem_cache_destroy(aufs_inode_cache);
    aufs_inode_cache = NULL;
}

void kmem_cache_destroy(struct kmem_cache *cachep);
```

rcu\_barrier - распространенный механизм синхронизации, он поставлен для безопасного освобождения памяти, для так называемых lock-free алгоритмов

kmem\_cache\_destroy удаляет SLAB аллокатор

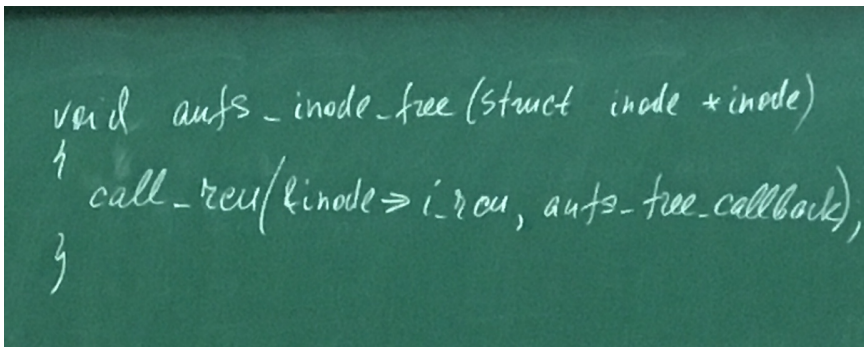
rcu\_barrier - блокирующая функция, ждет пока все отложенные действия над защищенными rcu данными завершатся, после этого возвращает управление коду, который ее вызвал.

```
struct inode *aufs_inode_alloc(struct super_block *sb)
{
    struct aufs_inode *cont i = (struct aufs_inode *) kmem_cache_alloc(aufs_inode_cache,
    if(!i) return NULL;
    return &i->ai_inode;
}

void kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags);
```

ГЛАВ. указ. то куда брать память из RAM а не

Функция возвращает объект из кэша. Если кэш пуст, то функция может вызвать функцию cache\_alloc\_refill(), чтобы добавить в кэш память



```
void aufs_inode_free(struct inode *inode)
{
    call_rcu(&inode->i_rcu, aufs_free_callback);
}
```

Освобождение  
inode'a с  
использованием rcu.

Блокировки это зло.

Проблема lock-free  
алгоритмов  
заключается в том,

что без блокировок нет гарантий, что мы получим правильный результат, так как у нас нет гарантий, что этот же объект не используется каким-то другим параллельным потоком выполнения. Поэтому, lock-free алгоритмам приходится заботиться о безопасном освобождении памяти. Именно такие средства предоставляет rcu.

Rcu-call откладывает выполнение функции aufs\_free\_callback

```
#cat /proc/slabinfo
```

```
...
```

```
inode_cache 7005 14792 480 1598 1849 1
```

```
dentry_cache 5469 5880 128 183 196 1
```

Описание числовых полей на записывыше слева направо.

Число активных объектов, общее число объектов, доступные объекты, размер каждого объекта в байтах, число страниц с относящихся по крайней мере к одному объекту, количество страниц, выделенных слабу.

-----



## Прерывания.

Шоу, “Логическое проектирование операционных систем”, там указана

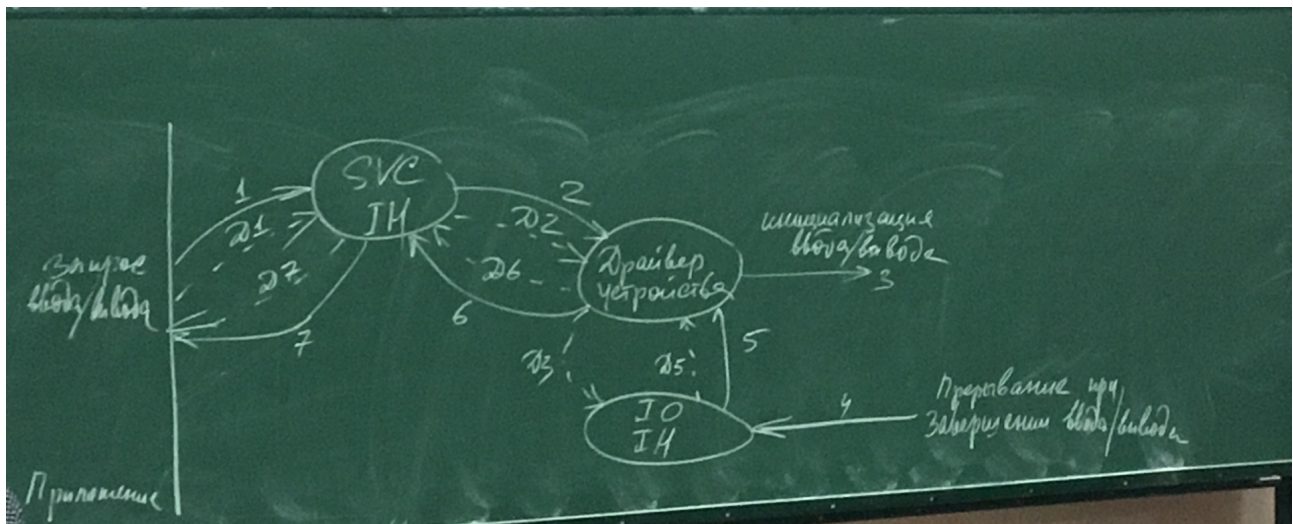


схема прерываний, приведенная ниже.

SVC - supervisor call, операционная система с стадии выполнения  
IH - interrupt handler

При обработке ввода/вывода система переводится в режим ядра. Ни одна ос не позволяет приложениям обращаться к устройствам ввода/вывода напрямую. В противном случае, такую систему нельзя было бы защитить.

Если мы вызываем какую-то библиотечную функцию ввода вывода, то в ее тексте обязательно будет системный вызов, выполняя которую система перейдет в режим ядра и начнет выполняться код ядра. Процесс в синхронном блокирующем вводе/выводе будет блокирован.

Речь в данном случае идет о синхронном блокирующем вводе/выводе. Блокировка процесса - непростой процесс для системы. Из диаграммы видно, что задействуются разные части кода ядра системы.

Безусловно, операционная система имеет соответствующие модули, которые часто называются подсистемой ввода/вывода. Эта подсистема определяет какой драйвер использовать, чтобы дать данные(?) в понятном виде для пользователя.

На каждом этапе данные переформатируются, то есть они получают определенный вид. Драйвер получает определенный формат данных, который должен будет преобразовать команду, которую получит контроллер устройства, который начнет управление устройством по

получении этой команды. При получении сигнала прерывания по окончании ввода вывода. Соответственно, сигнал прерывания, который приходит на контроллер может быть замаскирован, то такой сигнал будет игнорироваться. Если не маскирован, то контроллер сформирует сигнал прерывания, который поступит на один из входов процессора. В конце цикла выполнения каждой команды, контроллер проверяет наличие сигнала прерывания на этой своей выделенной ножке (входе). Если сигнал поступил, то процессор переходит на выполнение обработчика прерывания и это обработчик аппаратного прерывания. IO IN на схеме. Все такие прерывания приходят от устройств - аппаратные, плюс прерывания от системного таймера тоже аппаратные. Все аппаратные прерывания выполняются на очень высоком уровне приоритетов, фактически когда они выполняются никакая другая работа в системе выполняться не может, пока работа прерывания не будет завершена. С точки зрения системы это нехорошо, поэтому *аппаратное прерывание делится на две части*. В UNIX/LINUX их принято называть **top half** и **bottom half** - верхние и нижние половины. В Windows эта же идея реализована с помощью `deferred procedure call` - отложенный вызов процедуры.

Аппаратные прерывания принято делить на быстрые и медленные. Быстрое не делится на две половины, выполняется как одно целое. В современном LINUX быстрым прерыванием являются только прерывания от системного таймера. Нижняя половина прерывания выполняется в UNIX/LINUX как отложенное действие. То есть, фактически аппаратным прерыванием является верхняя половина, задачи которой следующие: получение данных из регистра устройства и помещение их в буфер ядра. Аппаратное прерывание, как правило, завершается инициализацией отложенных действий, например путем постановки соответствующей нижней половины в очередь на выполнение, для этого например вызывается функция `scheduler` (планирования). Если вернуться к диаграмме выше, то в любом случае, или команда чтения, или записи, все равно необходимо передать информацию об успешном завершении операции вывода. Соответственно в любом случае процесс должен получить информацию об успешном завершении операции. 5 стрелка и стрелка D5 с помощью соответствующих функций драйвера, драйвер должен инициировать передачу данных из буфера ядра в буфер приложения, а для этого процесс должен быть разблокирован. Обеспечение ввода вывода - одна из самых сложных задач, которые решает операционная система.