В отличие от обьекта, суперблок и айнод не сопоставляется …

Вирт файл система сохдает обьект на лету, из строкового представления пути к файлу pass name, как все структуры ядра.ю структура denntry претерпевает изменения в новых версиях ядра. Мы должны сопоставлять структуры с версией ядра.

Dentry – структура работает с именами файлов. В линукс/юникс начинается с корневого каталога полное имя. Каждая поддиректория в юниукс/линукс это специальный файл. Каталоги представляются как файлы, но это специальные файлы, с которыми м не можем работать, на них определены спец действия, у этих файлов свои спец задачи – обеспечить возможность по символьному имени файла обеспечить доступ к физическому имени. Struct inode хранит адрес/путь – строка символов, которая разбирается, но по этой строке/пути надо в конечном итоге получить номер айнода, то есть обьект дентри это определенный компонент пути(файл). Причем все обекты дентри это компонетнты пути, включая обычные файлы. Компоненты пути могут включать в себя точки монтирования. Поскольку элементы объектов каталога не хранятся на физическом носителе, например на диске, структура struct dentry не имеет флагов, которые указывали бы на то, что объект был изменен, или как принято у юникс, объект dirty, потому не надо переносить на диск, а отредактированный айнод надо сопоставить с ним.

Struct dentry – код из ядра версии 3.14

На экзамен можно приносить структуры, просто голые структуры. Смысл всех полей надо знать и опнимать.

Struct dentry

{  
/\*RCU lookup touched fields\*/  
Unsigned int d\_flags; /\*protected by d\_lock\*/  
Seqcount\_t d\_seq; /\*per dentry seqlock\*/  
Struct hlist\_bl\_node d\_hash; /\*list of hash table entries\*/  
Struct dentry \*d\_parent;  
Struct qstr d\_name; /\*dentry name\*/  
struct inode \*d\_inode; /\*связанный inode, если – NVLL, то neqabive\*/  
unsigned char d\_iname[DNAME\_INLINE\_LEN]; //короткое имя  
…  
const struct dentry\_operations \*d\_op;  
struct super\_block \*d\_sb; /\*корневой каталог(root) дерева каталогов ФС\*/  
unsigned long d\_time;  
void \*d\_isdata; /\*данные спец. Для ФС\*/  
struct list\_head d\_lru/\*список LRU\*/  
…  
struct list\_head d\_subdirs;  
};

3 состояния из dentry:

* Used
* Unused
* Negative

//RCU read-copy-undate – механизм синхронизации, имеется в виду взаимоисключение(которое обеспечивает монопольный доступ) при доступе к определенным полям структуры. Синхронизация предполагает, что один процесс заинтересован в действиях другого процесса.

//фотки в 14.31

Dentry создается на лету, кешируется. Когда витрульаная ФС прошла и для каждого элемента прошла по пути, по полному имени файла, и для каждого элемента пути создан обект дентри, причем путь пройден полностью, то это расточительно всю эту работу выбросить. Очевидно, что в системе так не делается. В результате разбора пути, информация собирается в дентри кеше, или dcache, список из использоемыз дентри , связ с inode полем. Поскольку отдельно взятый inode может иметь много оюьектов dentry, связанных с этим inode, кореспондирующим этот inode.

Двусвязный список LRU – это список не используемых объектов dentry, отсортирован по времени. Соответственно в начале будет самый новый объект дентри, удаление элементов выполняется по LRU.

Третий список hash\_table хэширует функции для быстрого определения заданного пути в ассоциированный объект дентри. Hash table реализован в виде dentry\_hash\_table array. Каждый элемент является указателем на список хэшированных по тому же самому имени.

Struct dentry\_operations  
{  
 int(\*d\_revalidate)(struct dentry \*, unsigned int);  
 int(\*d\_hash)(const struct dentry \*, struct qstr \*);  
 int(\*d\_compare)(const struct dentry \*, const struct dentry \*, unsigned int, const char \*, const struct qstr \*);  
 int(\*d\_delete)(const struct dentry \*);  
 int(\*d\_release)(struct dentry \*);  
 …  
 void(\*d\_input)(struct dentry \*, struct inode \*);  
 …  
}

Функция работает, если заданный дентри обект сущестует. Вызывается эта функция, чтобы подготовить использование в дентри кеше. Большинство систем устанавливает его в 0, так как обыно все доступно. Функция д\_кеш создает значение величину кеш для заданного объекта дентри. ФС вызывает эту функцию, чтобы добавтть дентри в хэш тейбл. Компэр вызывается для сравнения двух имени файлов. Большиснтво ФС определяют ее как default, которая выполняет простое сравнение строк. Функция непосредственно связана с дентри. Функция дэлит вызывается, когда у соответвующего элемента дентри коунт становится равным нулю(д\_коунт в коде выше нет).

Айнод кеш представляет из себя:

* Глобальный хэш-массив inode\_hashtable, в котором каждый айнод хэшируется по указателю на суперрблок и к 32разрядному номер айнод. При отсутсвии блока, айнод добавляется к свифту анонимных айнодов(сокеты, созданные вызовом sock\_alloc() )
* Глобальный список inode\_in\_use. В этом списке сод допустимые айноды с (i\_count > 0 и i\_nlink > 0). В этот же список хэшируются вновь созданные айноды, объект айнод
* Глобальный список inode\_unused, кот допускает i\_count = 0
* Список для каждого суперблока (sb -> s\_dirty), кот сод айноды с i\_count>0, i\_nlink>0, состояние dirty, то есть измененные айноды
* SLAB cache, кот называется inode\_cachep. Айноды могут создаваться, изыматься из лап хэша. Айнод может находиться только в одном из этих списков.

SLAB cache – слаб(бревно) распределение было введено Джефом Бронвиком(Jeff Bonwick) для опер сист SunOS

SLAB аллокатор – хэширующий аллокатор, позволяющий выделять блоки памяти одного размера. Слаб распределение стало использоваться для упрощения управления памятью и ее выделения. Дело в том, что в ядре значительные объемы памяти выделяются на гор объем объектов(дескрипторы файлов, айноды). Идея бронвика позируется на том, что колво времение, необх для инициализации обычного объекта в ядре превышает кол-во времени, кот необходимо для его выделения и освобождения. Его идея заключается в том, чтовместо того, чтобы возвращать память системе, оставлять ее в проинициализированном состоянии, расчитывая на то, что она будет позже использоваться для тех же целей. Т.е. если надо выделить участок памяти для айнод, можем воспользоваться уже использованным участком для айнод.

Например, если память выделена для мьютикс, то его надо выполнить 1 раз, тогда память выполняется впервые, в послед распределения памяти не нужны, так как она уже выделена, а результате уже сделанной инициализации и последующего освобождения.

//фото в 15.24

Best\_fit – самый подходящий