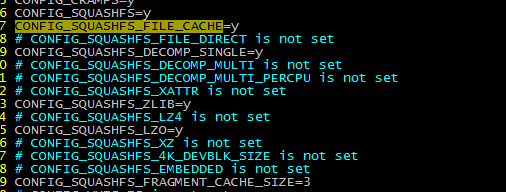
## squashfs\_fill\_super

struct squashfs\_super\_block，位于第一个扇区。非压缩

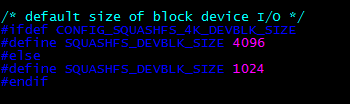












struct file\_system\_type squashfs\_fs\_type = {

.owner = THIS\_MODULE,

.mount = squashfs\_mount,

};

struct squashfs\_sb\_info {

struct squashfs\_stream \*stream;

};

squashfs\_mount->

squashfs\_fill\_super(sb,data,silent)

{

struct squashfs\_super\_block \*sblk = NULL;

struct squashfs\_sb\_info \*msblk;

unsigned short flags;

u64 next\_table;

long long root\_inode;

struct inode \*root;

sb->s\_fs\_info = kzalloc(sizeof(\*msblk),GFP\_KERNEL);

msbk= sb->s\_fs\_info;

msblk->devblksize = sb\_min\_blocksize(sb,SQUASHFS\_DEVBLK\_SIZE);

msblk->devblksize\_log2 = ffz(~msblk->devblksize);

/\*

sblk在squashfs\_read\_table中分配。

\*/

sblk = squashfs\_read\_table(sb,SQUASHFS\_START,sizeof(\*sblk));

sb->s\_magic = le32\_to\_cpu(sblk->s\_magic);

if(sb->s\_magic != SQUASHFS\_MAGIC) {

}

msblk->decompressor = supported\_squashfs\_filesystem(

le16\_to\_cpu(sblk->s\_major),

le16\_to\_cpu(sblk->s\_minor),

le16\_to\_cpu(sblk->compression));

msblk->bytes\_used = le64\_to\_cpu(sblk->bytes\_used);

if(msblk->bytes\_used <0 || msblk->bytes\_used >i\_size\_read())

{

}

flags = le16\_to\_cpu(sblk->flags);

msblk->block\_size =le32\_to\_cpu(sblk->block\_size);

msblk->block\_log = le16\_to\_cpu(sblk->block\_log);

root\_inode = le64\_to\_cpu(sblk->root\_inode);

if(SQUASHFS\_INODE\_OFFSET(root\_inode)>SQUASHFS\_METADATA\_SIZE)

goto failed;

msblk->inode\_table = le64\_to\_cpu(sblk->inode\_table\_start);

msblk->directory\_table =le64\_to\_cpu(sblk->directory\_table\_start);

msblk->inodes = le32\_to\_cpu(sblk->inodes);

sb->s\_op = &squashfs\_super\_ops;

sb->s\_flags |= MS\_RDONLY;

sb->s\_maxbytes = MAX\_LFS\_FILESIZE;

msblk->block\_cache = squashfs\_cache\_init(“metadata”,SQUASHFS\_CACHED\_BLKS,SQUASHFS\_METADATA\_SIZE);

msblk->read\_page = squashfs\_cache\_init(“data”,squashfs\_max\_decompressors(),msblk->block\_size);

msblk->stream = squashfs\_decompressor\_setup(sb,flags);

sb->s\_xattr = squashfs\_xattr\_handlers;

xattr\_id\_table\_start = le64\_to\_cpu(sblk->xattr\_id\_table\_start);

/\*

xattr，如果离线制作是镜像支持，那么linux端的一定要支持。

如果离线制作不支持，则linux端无所谓。

\*/

if(xattr\_id\_table\_start ==SQUASHFS\_INVALID\_BLK) {

next\_table = msblk->bytes\_used;

goto allocate\_id\_index\_table;

}

msblk->xattr\_id\_table = squashfs\_read\_xattr\_id\_table

(sb,xattr\_id\_table\_start,&msblk->xattr\_table,&msblk->xattr\_ids);

/\*

squashfs文件系统制作时必须保持一致

\*/

if(IS\_ERR(msblk->xattr\_id\_table))

{

挂载失败；

}

next\_table = msblk->xattr\_table;

allocate\_id\_index\_table:

msblk->id\_table = squashfs\_read\_id\_index\_table

(sb,le64\_to\_cpu(sblk->id\_table\_start),next\_table,le16\_to\_cpu(sblk->no\_ids));

next\_table = le64\_to\_cpu(msblk->id\_table[0]);

lookup\_table\_start = le64\_to\_cpu(sblk->lookup\_table\_start);

if(lookup\_table\_start ==SQUASHFS\_INVALID\_BLK)

goto handle\_fragments;

msblk->inode\_lookup\_table = squashfs\_read\_inode\_lookup\_table

(sb,lookup\_table\_start,next\_table,msblk->inodes);

next\_table = le64\_to\_cpu(msblk->inode\_lookup\_table[0]);

sb->s\_export\_op =&squashfs\_export\_ops;

handle\_fragments:

fragments = le32\_to\_cpu(sblk->fragments);

if(fragments ==0)

goto check\_directory\_table;

msblk->fragment\_cache = squashfs\_cache\_init

(“fragments”,SQUASHFS\_CACHED\_FRAGMENTS,msblk->block\_size);

msblk->fragment\_index = squashfs\_read\_fragment\_index\_table

(sb,le64\_to\_cpu(sblk->fragment\_table\_start),next\_table,fragments);

next\_table = le64\_to\_cpu(msbk->fragment\_index[0]);

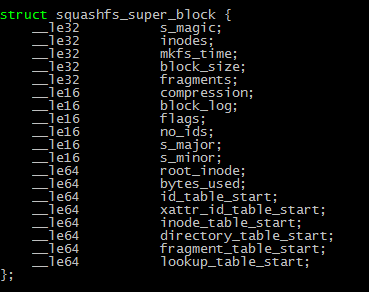
check\_directory\_table:

root =new\_inode(sb);

err = squashfs\_read\_inode(root,root\_inode);

}

## squashfs\_read\_table (squashfs\_super\_block)



读取squashfs的开头的未压缩的struct squashfs\_super\_block结构。

squashfs\_read\_table(sb,SQUASHFS\_START,sizeof(\*sblk)) {

struct squashfs\_page\_actor \*actor;

u64 block = SQUASHFS\_START;

int length = sizeof(\*sblk);

int res;

int pages =(length +PAGE\_SIZE-1) >>PAGE\_SHIFT;

void \*table ,\*buffer.\*\*data;

data = kcalloc(pages,sizeof(void \*),GFP\_KERNEL);

buffer = table = kmalloc(length,GFP\_KERNEL);

/\*

page\_actor decompressing into an intermediate buffer.

for decompressing directly into the page cache

\*/

actor = squashfs\_page\_actor\_init(data,pages,length);

for(i=0;i<pages;i++,buffer+=PAGE\_SIZE)

data[i] = buffer;

res = squashfs\_read\_data(sb,block,length | SQUASHFS\_COMPRESSED\_BIT\_BLOCK,NULL，actor);

/\*

释放二级指针第一级

\*/

kfree(data);

/\*

释放squashfs\_page\_actor\_init中分配的actor

\*/

kfree(actor);

return table;

}

## page\_actor

squashfs分为CONFIG\_SQUASHFS\_FILE\_DIRECT，解压数据直接到page\_cache，可以减少拷贝和减轻lock contention.

在squashfs/page\_actor.c中squashfs\_page\_actor\_init的实现是直接拷贝，在squashfs\_page\_actor\_init中squashfs\_page\_actor\_init的实现是存在中间buffer

squashfs\_page\_actor\_init(data,pages,length)

{

void \*\*page =data;

struct squashfs\_page\_actor \*actor = kmalloc(sizeof(\*actor),GFP\_KERNEL);

actor->length = length;

/\*

page不是指的page，而是buffer

\*/

actor->page =page;

actor->pages = pages;

actor->next\_page =0;

return actor;

}

## squashfs\_read\_data

squashfs\_read\_data(sb,block,length|SQUASHFS\_COMPRESSED\_BIT\_BLOCK

,NULL，actor)

{

struct buffer\_head \*\*bh;

struct squashfs\_page\_actor \*output = actor;

u64 \*next\_index = NULL;

int length = length | SQUASHFS\_COMPRESSED\_BIT\_BLOCK;

u64 index= block;

int compressed ;

struct squashfs\_sb\_info \*msblk = sb->s\_fs\_info;

int offset = index &((1<<msblk->devblksize\_log2)-1);

int bytes,b=0,k=0;

bh = kcalloc((output->length+msblk->devblksize-1)>>msblk->devblksize\_log2)+1,sizeof(\*bh),GFP\_KERNEL);

if(length)

{

/\*

data block

\*/

bytes = -offset;

compressed = SQUASHFS\_COMPRESSED\_BLOCK(length);

length = SQUASHFS\_COMPRESSED\_SIZE\_BLOCK(length);

for(b=0;bytes<length;b++,cur\_index++) {

/\*

sb\_getblk会通过sb->s\_bdev来进行读操作

\*/

bh[b] = sb\_getblk(sb,cur\_index);

bytes+=msblk->devblksize;

}

ll\_rw\_block(REQ\_OP\_READ,0,b,bh);

}else

{

/\*

medadata block

\*/

bh[0] = get\_block\_length(sb,&cur\_index,&offset,&length);

b = 1;

bytes = msblk->devblksize – offset;

compressed = SQUASHFS\_COMPRESSED(length);

length = SQUASHFS\_COMPRESSED\_SIZE(length);

if(next\_index)

\*next\_index = index + length+2;

for(;bytes < length;b++)

{

bh[b] = sb\_getblk(sb,++cur\_index);

bytes += msblk->devblksize;

}

ll\_rw\_block(SEQ\_OP\_READ,0,b-1,bh+1);

}

for(i=0;i<b;i++) {

wait\_on\_buffer(bh[i]);

if(!buffer\_uptodate(bh[i]))

goto block\_release;

}

if(compressed) {

length = squashfs\_decompress(msblk,bh,b,offset,length,output);

}else {

/\*

block is uncompressed

\*/

int in,pg\_offset =0;

void \*data =squashfs\_first\_page(output);

for(bytes =length;k<b;k++)

{

in = min(bytes,msblk->devblksize-offset);

bytes-=in;

while(in) {

avail =min\_t(int,PAGE\_SIZE-pg\_offset);

memcpy(data+pg\_offset,bh[k]->b\_data+offset,avail);

in -= avail;

pg\_offset+=avail;

offset+=avail;

}

offset =0;

put\_bh(bh[k]);

}

/\*

非direct实现为空

\*/

squashfs\_finish\_page(page);

}

kfree(bh);

return length;

}

squashfs\_first\_page(output)

{

actor->next\_page =1;

return actor->page[0];

}

/\*

read the metadata block length,this is stored in the first two bytes of the metadata block

\*/

get\_block\_length(sb,&cur\_index,&offset,&length)

{

struct buffer\_head \*bh;

bh = sb\_bread(sb,\*cur\_index);

/\*

如果metadata的第一个字节在block的最后一个字节，则需要读出下一个字节才能获取长度

\*/

if(msblk->devblksize - \*offset == 1)

{

\*length = (unsigned char)bh->b\_data[\*offset];

put\_bh(bh)；

bh = sb\_bread(sb,++(\*cur\_index));

\*length |= (unsigned char) bh->b\_data[0] <<8;

\*offset =1;

}else {

\*length = (unsigned char)bh->b\_data[\*offset] |

(unsigned char)bh->b\_data[\*offset+1]<<8;

\*offset +=2;

if(\*offset == msblk->devblksize) {

if(\*offset == msblk->devblksize) {

put\_bh(bh);

bh = sb\_bread(sb,++(\*cur\_index));

\*offset =0;

}

}

}

return bh;

}

## supported\_squashfs\_filesystem

supported\_squahfs\_filesystem(major,minor,id)

{

struct squashfs\_decompressor \*decompressor;

if(major<SQUASHFS\_MAJOR ) {

}else if(major > SQUASHFS\_MAJOR || minor < SQUASHFS\_MINOR)

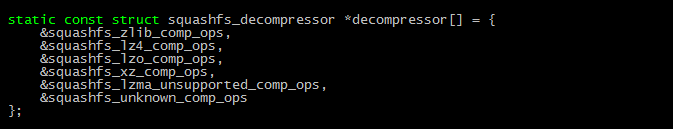
{

}

decompressor = squa shfs\_lookup\_decompressor(id);

return decompressor;

}



squashfs\_lookup\_decompressor(id)

{

int i;

for(i=0;decompressor[i]->id;i++)

if(id==decompressor[i]->id)

break;

return decompressor[i];

}

squashfs\_decompressor\_setup(sb,flags)

{

void \*stream;

void \*comp\_opts = get\_comp\_opts(sb,flags);

/\*

squashfs\_decompressor\_setup实现有多线程、单线程和多cpu的，主要是调用decompressor的回调init

\*/

stream = squashfs\_decompressor\_setup(msblk,comp\_opts);

}

get\_comp\_opts(sb,flags)

{

void \*buffer =NULL;

void \*comp\_opts;

int length =0;

/\*

SQUASHFS\_COMP\_OPTS,如果flags的第10bit置位

\*/

if(SQUASHFS\_COMP\_OPTS(flags)) {

buffer = kmalloc(PAGE\_SIZE,GFP\_KERNEL);

actor = squashfs\_page\_actor\_init(&buffer,1,0);

/\*

再次读一遍struct squashfs\_super\_block

\*/

length = squashfs\_read\_data(sb,sizeof(struct squashfs\_super\_block),

0,NULL,actor);

}

/\*

查看decompressor的回调是否有comp\_opts，如果没有，则返回NULL

\*/

comp\_opts = squashfs\_comp\_opts(msblk,buffer,length);

kfree(actor);

kfree(buffer);

return comp\_opts;

}

squashfs\_decompress(msblk,bh,b,offset,length,output) {

int res;

struct squashfs\_stream \*stream = msblk->stream;

res = msblk->decompressor->decompress(

msblk,stream->stream,bh,b,offset,length,output);

}

## squashfs\_cache\_init

#define SQUASHFS\_METADATA\_SIZE 8192

#define SQUASHFS\_CACHED\_BLKS 8

#define SQUASHFS\_INVALID\_BLK (-1LL)

struct squashfs\_cache\_entry {

struct squashfs\_page\_actor \*actor;

u64 block;

int num\_waiters;

wait\_queue\_head\_t wait\_queue;

int unused;

int refcount;

};

struct squashfs\_cache {

struct squashfs\_cache\_entry \*entry;

void \*\*data;

};

squashfs\_cache\_init(“metadata”,

SQUASHFS\_CACHED\_BLKS,

SQUASHFS\_METADATA\_SIZE)

{

int entries = SQUASHFS\_CACHED\_BLKS = 8;

int block\_size = SQUASHFS\_METADATA\_SIZE = 8192;

int i=0;

struct squashfs\_cache \*cache = kzalloc(sizeof(\*cache),GFP\_KERNEL);

cache->entry = kcalloc(entries,sizeof(\*(cache->entry)),GFP\_KERNEL);

cache->curr\_blk =0;

cache->next\_blk =0;

cache->block\_size = block\_size;

cache->entries = entries;

cache->pages = block\_size >> PAGE\_SHIFT;

cache->pages = cache->pages?cache->pages:1;

cache->name = name;

for(i=0;i<entries;i++) {

struct squashfs\_cache\_entry \*entry = &cache->entry[i];

entry->cache = cache;

entry->block = SQUASHFS\_INVALID\_BLK =-1；

entry->data = kcalloc(cache->pages,sizeof(void \*),GFP\_KERNEL);

for(int j=0;j<cache->pages;j++)

{

entry->data[j] = kmalloc(PAGE\_SIZE,GFP\_KERNEL);

}

entry->actor = squashfs\_page\_actor\_init(entry->data,cache->pages,0);

}

return cache;

}

## xattr

如果CONFIG\_SQUASHFS\_XATTR没有定义，

squashfs\_read\_xattr\_id\_table

(sb,xattr\_id\_table\_start,&msblk->xattr\_table,&msblk->xattr\_ids)

{

u64 start = xattr\_id\_table\_start;

u64 \*xattr\_table\_start = &msblk->xattr\_table;

\*xattr\_table\_start = start;

return ERR\_PTR(-ENOTSUPP);

}

## idtable

squashfs\_read\_id\_index\_table

(sb,le64\_to\_cpu(sblk->id\_table\_start),next\_table,le16\_to\_cpu(sblk->no\_ids))

{

unsigned short no\_ids = le16\_to\_cpu(sblk->no\_ids);

unsigned int length = SQUASHFS\_ID\_BLOCK\_BYTES(no\_ids);

\_\_le64 \*table;

if(id\_table\_start+length > next\_table)

return ERR\_PTR(-EINVAL);

table = squashfs\_read\_table(sb,id\_table\_start,length);

return table;

}

## inode\_lookup\_table

squashfs\_read\_inode\_lookup\_table

(sb,lookup\_table\_start,next\_table,msblk->inodes) {

unsigned int length = SQUASHFS\_LOOKUP\_BLOCK\_BYTES(inodes);

\_\_le64 \*table;

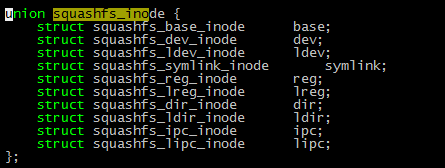
table = squashfs\_read\_table(sb,lookup\_table\_start,length);

return table;

}

## fragments

## squashfs\_read\_inode



struct squashfs\_base\_inode {

le16 inode\_type;

le16 mode;

le16 uid;

le16 guid;

le32 mtime;

le32 inode\_number;

};



squashfs\_read\_inode(root,root\_inode)

{

long long ino = root\_inode;

struct inode \*inode = root;

union squashfs\_inode squashfs\_ino;

struct squashfs\_base\_inode \*sqshb\_ino =&squashfs\_ino.base;

u64 block = SQUASHFS\_INODE\_BLK(ino) + msblk->inode\_table;

int offset = SQUASHFS\_INODE\_OFFSET(ino);

squashfs\_read\_metadata(sb,sqshb\_ino,&block,&offset,sizeof(\*sqshb\_ino));

}

squashfs\_read\_metadata(sb,sqshb\_ino,&block,&offset,sizeof(\*sqshb\_ino))

{

struct squashfs\_sb\_info \*msblk = sb->sb\_fs\_info;

int length = sizeof(\*sqshb\_ino);

int res = length,bytes;

struct squashfs\_cache\_entry \*entry;

void \*buffer = sqshb\_ino;

while(length) {

entry = squashfs\_cache\_get(sb,msblk->block\_cache,\*block,0);

bytes = squashfs\_copy\_data(buffer,entry,\*offset,length);

if(buffer)

buffer+=bytes;

length -=bytes;

\*offset +=bytes;

if(\*offset == entry->length)

{

\*block = entry->next\_index;

\*offset = 0;

}

}

}

squashfs\_cache\_get(sb,msblk->block\_cache,\*block,0)

{

struct squashfs\_cache \*cache = msblk->block\_cache;

u64 block = \*block;

int length =0;

int i,n;

while(1)

{

/\*

i只是初始会被赋值成curr\_blk，n是一个遍历步骤，i才是目的；

\*/

for(i=cache->curr\_blk,n=0;n<cache->entries;n++)

{

if(cache->entry[i].block ==block)

{

cache->curr\_blk =i;

break;

}

i = (i+1)%cache->entries;

}

if(n==cache->entries) {

/\*

还要等待cache

\*/

if(cache->unused ==0)

{

cache->num\_waiters++;

wait\_event(cache->wait\_queue,cache->unused);

cache->num\_waiteds--;

}

}

i = cache->next\_blk;

for(n=0;n<cache->entries;n++) {

if(cache->entry[i].refcount ==0)

break;

i = (i+1)%cache->entries;

}

cache->next\_blk = (i+1)%cache->entries;

entry = &cache->entry[i];

cache->unused--;

entry->block = block;

entry->refcount =1;

entry->pending =1;

entry->num\_waiters =0;

entry->error =0;

entry->length = squashfs\_read\_data

(sb,block,length,&entry->next\_index,entry->actor);

}

}