文件系统

磁盘同步函数

1. Sync

只是将所有修改过的块缓冲区排入写队列，然后就返回，它并不等待实际写磁盘操作结束。通常称为update的系统守护进程会周期性地（一般每隔30秒）调用sync函数.

2. Fsync

只对由文件描述符filedes指定的单一文件起作用，并且等待写磁盘操作结束，然后返回。fsync还会同步文件的描述信息（metadata，包括size、访问时间st_atime & st_mtime等等）

3. Fdatasync

类似于fsync，但它只影响文件的数据部分。

文件系统的同步

同步就是将内存中的脏页写入磁盘，保证磁盘和物理页的内容一致。触发同步的时机:

1. 周期性的扫描
2. 脏页过多（延迟写的缓冲区写满了）
3. sync等函数调用

崩溃一致性

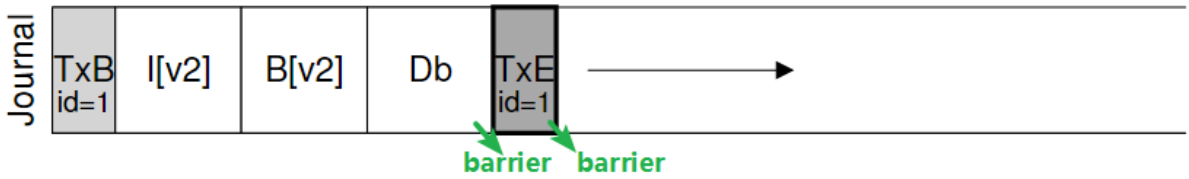
Shadow paging（CoW）

copy-on-write，存在双倍写问题。由于文件数据一般存储在树种，叶子节点的数据更改引起大量的copy-on-write。

> When updates have propagated to the top of the tree, a single write to the root of the tree commits all updates to “live” storage.

日志

journal，即write-ahead-logging。写文件要写bitmap+inode+filedata，为了一致性，在写前写日志。一段日志的写必须是一个事务，所以会用barrier保证。



然而，对于一个block数据的写可能会出现写了一部分的情况，这是**原子性问题**

写顺序

1. 写日志，包括TxB，修改后的data block
2. 提交日志，包括TxB
3. checkpoint，即将修改写到最终位置
4. 释放日志空间，使用**journal superblock**

Block resuse问题

即在metadata journaling下，如果先前有block被当作元数据写到journal，随后被delete，紧接有new file重用删除的block。replay过程中，可能会重新写该block，导致user data被改变。

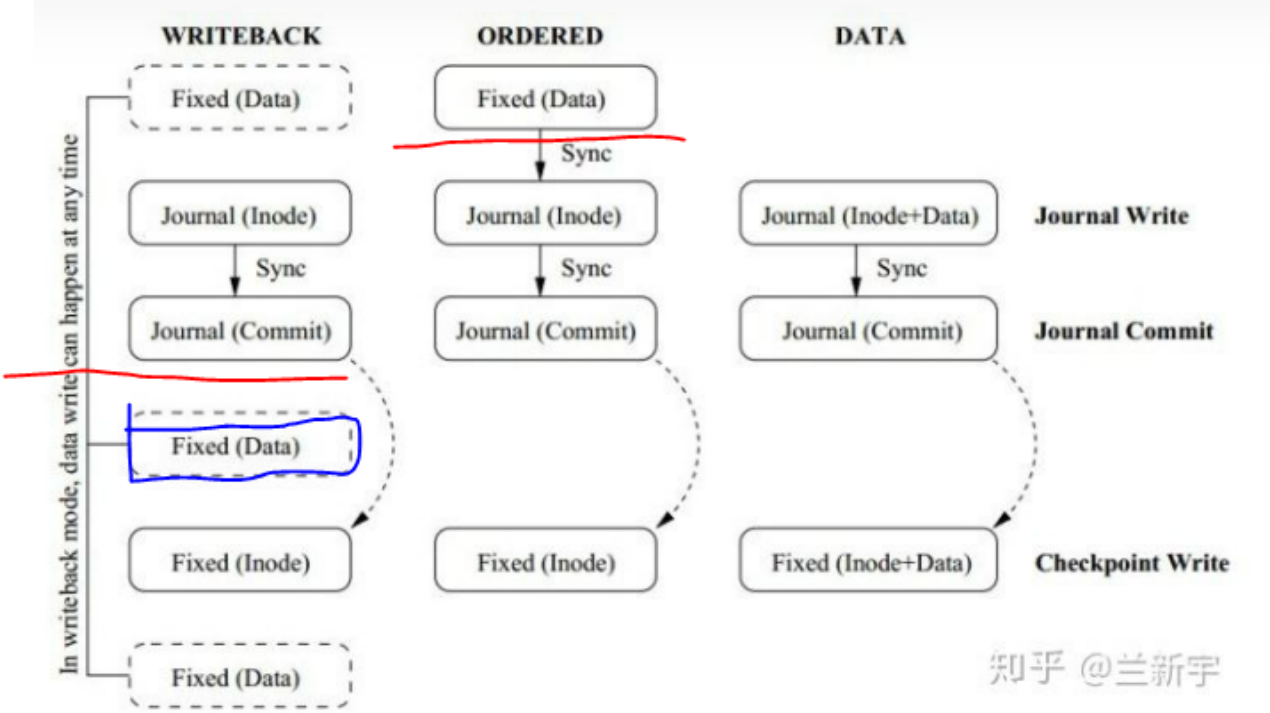
解决方法：

- 直到被删除的block从journal释放，才重用该block
- revoke标记的日志，不会被raplay

优化

1. 将多个journal聚合处理
2. 不对data做日志，即metadata journaling，更流行

问题：由于磁盘的out of order，user data的写可能发生在任何节点，即Writeback



Writeback：crash发生在红线处，会导致同一文件的meta data和user data不一致

ordered：crash发生红线处，造成一部分user data丢失，但不会造成不一致（？如果是就地更新呢

> 理解：这里的一致指的是文件系统角度的一致，在用户看来是不一致的

从左到右：**风险降低，性能损耗升高**

使用的最多的是ordered metadata journaling

3. **checksum**

保证每次日志写是一个事务，提交前需要等待。使用事务校验和，在恢复时可以检查该事务是否完整，因此可以同时写事务。

Timeline

issue表示可以同时开始

complete不表示完成顺序

虚线代表barrier

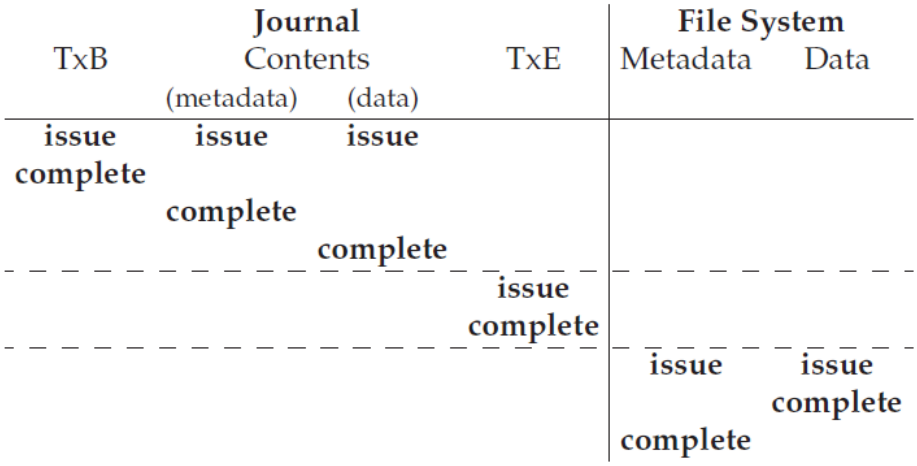


Figure 42.1: **Data Journaling Timeline**

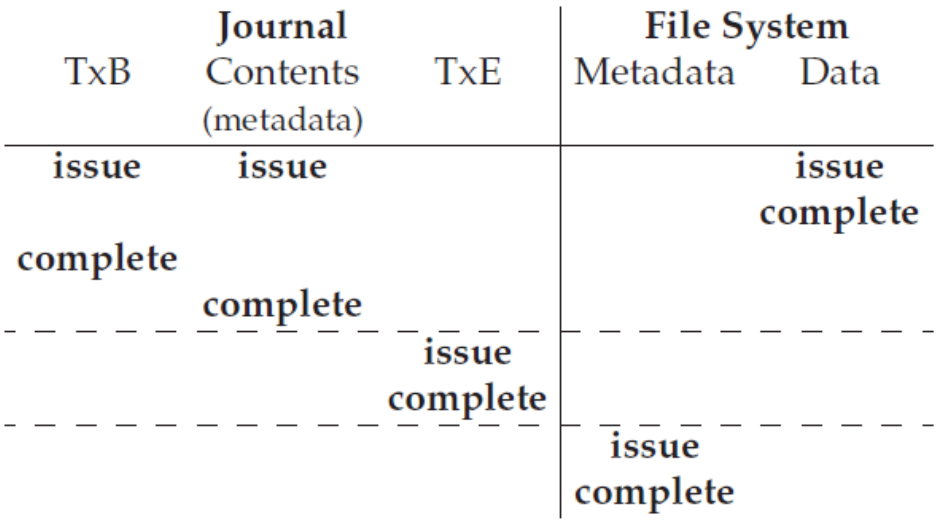


Figure 42.2: **Metadata Journaling Timeline**

Other

1. Soft Updates

仔细排序所有磁盘写，保证不会处于不一致状态

2. Back pointer

每个data block还有一个指回其inode的指针，只要双向指向，才能说明一致

一些文件系统

旧UNIX文件系统（不考虑磁盘特性）

很简单，但性能表现差：

数据分散保存，寻址代价大。一个逻辑上连续的文件分散到磁盘上存储

快速文件系统FFS（磁盘感知）

问题：传统FS空闲空间管理不行，有大量碎片，文件**散落**在磁盘中。寻道时间长，磁盘带宽利用不足

- 将磁盘分为多个柱面组(cylinder group)

柱面指多个磁盘面的同一磁道之和，柱面组呈现给上层文件系统即block groups

- 每个组包含fs的全部结构，位图用作空闲空间管理，每个组都有super block用作备份

- 放置策略，即将相关数据放在一个组

对于目录：选取最空闲的组

对于文件：数据块和inode同一组，同一目录的所有文件同一组（来源于常识，并非测试）

- 大文件分散到多个组中

放在一个group会占满空间，破环group的优势

简单的分散方法：目录的直接索引指向的数据块放到一个组，一级索引指向的数据块放到下一个最空闲的组

> 一个疑惑：blcok size被认为是transfer data和寻道时间的trade off ???

- 其他创新

对于小文件，在内存缓存到4KB再写入；否则，引入了大小为512B的sub-block

优化了磁盘布局，避免读取下一个block时错过，要等待转回来：

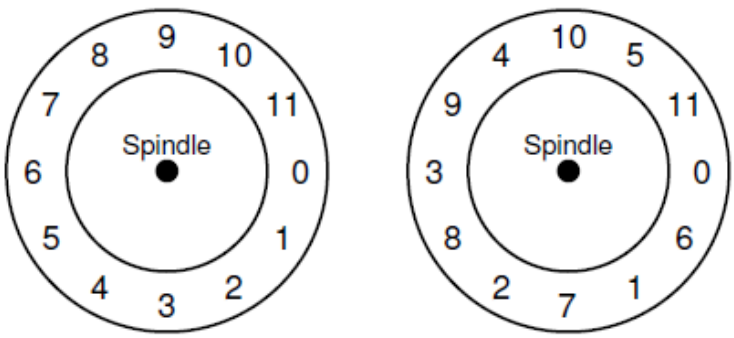


Figure 41.3: **FFS: Standard Versus Parameterized Placement**

> 现在更是引入了track buffer，以解决再次读取同一个block要等待的问题

支持符号链接，长文件名，rename等

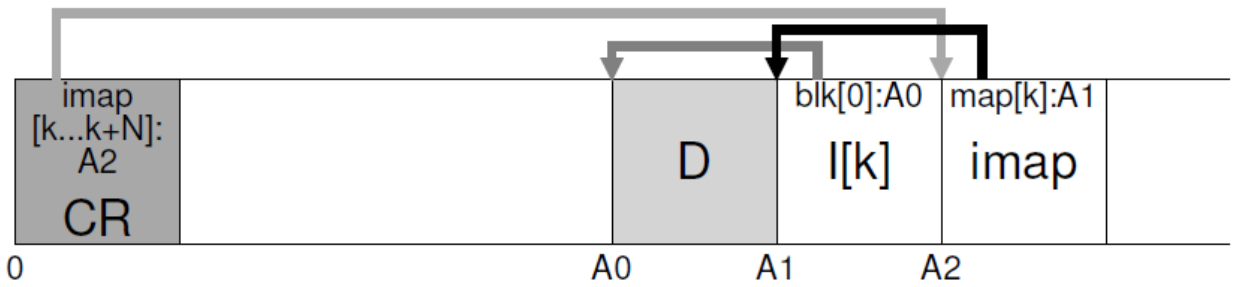
日志结构文件系统LFS

Data structure	Purpose	Location
Inode	Locates blocks of file, holds protection bits, modify time, etc.	Log
Inode map	Locates position of inode in log, holds time of last access plus version number.	Log
Indirect block	Locates blocks of large files.	Log
Segment summary	Identifies contents of segment (file number and offset for each block).	Log
Segment usage table	Counts live bytes still left in segments, stores last write time for data in segments.	Log
Superblock	Holds static configuration information such as number of segments and segment size.	Fixed
Checkpoint region	Locates blocks of inode map and segment usage table, identifies last checkpoint in log.	Fixed
Directory change log	Records directory operations to maintain consistency of reference counts in inodes.	Log

从不就地更新

大量写被组织为segment，然后写进磁盘，一个segment包含多个block

每次写：CR指向最新版本的imap



垃圾回收

segment也是垃圾回收的粒度

每个segment的头部有segment summary block，记录每一个block的inode number和offset

原论文使用segment usage table（CR指向），对于每个segment，记录有效字节数目，和最近修改时间

- 判断block的生命
  a. block位于地址A，在segment summary block中的inode number为N，偏移为T

process is shown here:

```
(N, T) = SegmentSummary[A];
inode  = Read(imap[N]);
if (inode[T] == A)
    // block D is alive
else
    // block D is garbage
```

b. 同时在imap和segment summary block中记录版本号，用于对比

故障恢复

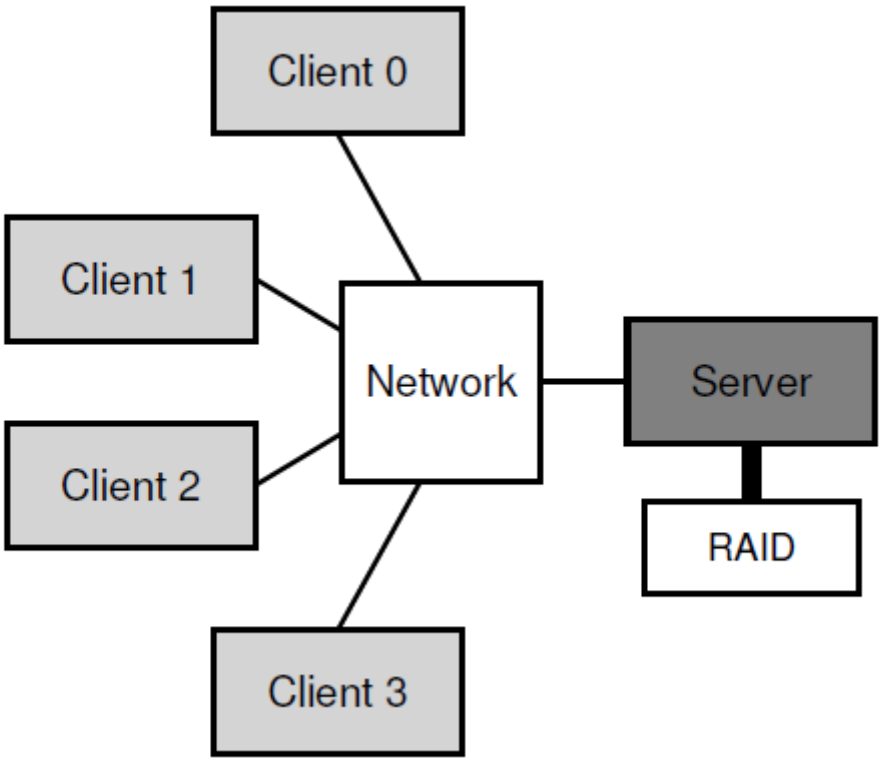
两个CR交替更新，更新间隔是30s

读CR指向的最新inode map，但未被CR记录的会丢失

roll forward:

根据segment summary block查看是否有有效block

分布式文件系统



Server：存储数据，server-side file system

Client：向server提出要求，client-side file system

特点：对于client的使用fs的文件系统来说，是透明的

Sun's Network File System(NFSv2)

NFS需要挂载到VFS下

1. 重点关注：simple and fast server crash recovery

如果client端使用文件描述符向server请求数据，那么如果server崩溃，重启后不知道该打开哪个文件

如果client打开文件后崩溃，由于没有发送close()给server，会导致文件一直打开

2. 主要思想：stateless，即server不存有关client的状态信息，但client存放完成一个请求所需的全部信息（以防server崩溃）

client获取file handle通过LOOKUP：client发送请求(directory file handle，file name)

WRITE/READ：发送（file handle，offset，number of bytes）

Client	Server
fd = open("/foo", ...); Send LOOKUP (rootdir FH, "foo")	
	Receive LOOKUP request look for "foo" in root dir return foo's FH + attributes
Receive LOOKUP reply allocate file desc in open file table store foo's FH in table store current file position (0) return file descriptor to application	
read(fd, buffer, MAX); Index into open file table with fd get NFS file handle (FH) use current file position as offset Send READ (FH, offset=0, count=MAX)	
	Receive READ request use FH to get volume/inode num read inode from disk (or cache) compute block location (using offset) read data from disk (or cache) return data to client
Receive READ reply update file position (+bytes read) set current file position = MAX return data/error code to app	
read(fd, buffer, MAX); Same except offset=MAX and set current file position = 2*MAX	

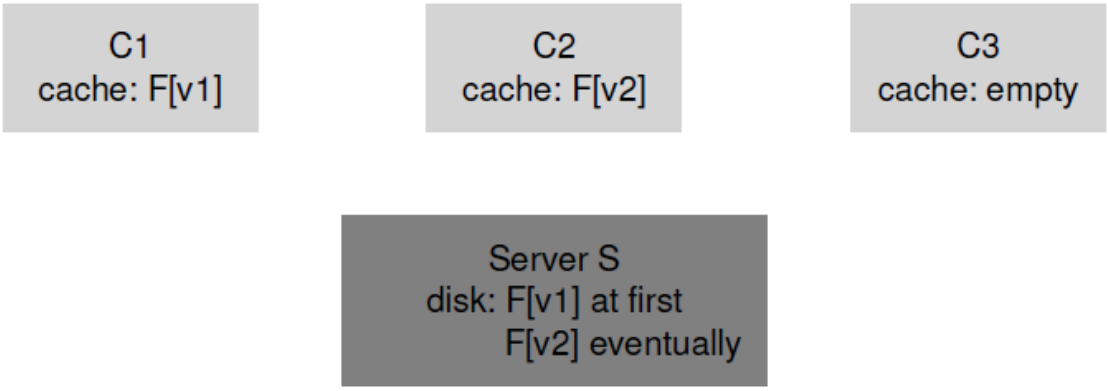
3. Server故障

stateless的特性保证，无论是server还是network故障，client简单地retry就可以了

4. Client-side Caching

缓存file block在内存中，同时可以作为写缓冲

5. Cache consistency problem



两个问题及对应解决方法

a. Update visibility，即C3什么时候能读到最新数据。方法是flush-on-close，当C2关闭F时，将更新提交给server
b. Stale cache，即C1的缓存已经失效。方法：当client使用cache，会发送GETATTR给server。带来的问题是，当只有一个client访问一个file时，会频繁地发送GETATTR给server。所以方法2是，在一段时间内认为cache没有失效，比如每隔3s向server询问（同样有弊端

6. Server's write

即使server端也会使用内存作为缓存，但只有当数据写入持久存储设备后，才返回成功给client。这导致写性能较差，进而写优化，比如：battery-backed memory和specifically designed to write to disk quickly

The Andrew File System(AFS)

主要目标：scale，而NFS中频繁的通信限制了server可以支持的clients数目

AFSv1

打开文件后，将整个文件内容缓存在本地disk中，关闭文件后，如果文件修改了则刷新回server

FETCH协议：client将整个文件路径发送给server-->问题：server花费太多时间在path walk上

在使用本地缓存的文件数据前，client向server发送请求确认是否已改变-->server需要回复太多请求

最终：server CPU成为瓶颈

AFSv2

1. server callback

由server通知client，文件数据已经改变

2. 修改FETCH

每次client将FID发送给server，并获取目录内容，长文件路径需要多次FETCH

Client (C_1)	Server
fd = open("/home/remzi/notes.txt", ...);	
Send Fetch (home FID, "remzi")	
	Receive Fetch request look for remzi in home dir establish callback(C_1) on remzi return remzi's content and FID
Receive Fetch reply write remzi to local disk cache record callback status of remzi	
Send Fetch (remzi FID, "notes.txt")	
	Receive Fetch request look for notes.txt in remzi dir establish callback(C_1) on notes.txt return notes.txt's content and FID
Receive Fetch reply write notes.txt to local disk cache record callback status of notes.txt local `open()` of cached notes.txt return file descriptor to application	
read(fd, buffer, MAX);	
perform local `read()` on cached copy	
close(fd);	
do local `close()` on cached copy if file has changed, flush to server	
fd = open("/home/remzi/notes.txt", ...);	

Foreach dir (home, remzi)
if (callback(dir) == VALID)
use local copy for lookup(dir)
else
Fetch (as above)
if (callback(notes.txt) == VALID)
open local cached copy
return file descriptor to it
else
Fetch (as above) then open and return fd	

3. cache consistency

Update visibility：同样是关闭文件后刷新给server，不同的是，文件的更新对同一client的进程立即可见

Cache staleness：server发送callback

Client$_1$			Client$_2$		Server	*Comments*
P_1	P_2	Cache	P_3	Cache	Disk	
open(F)		-		-	-	File created
write(A)		A		-	-	
close()		A		-	A	
	open(F)	A		-	A	
	read() → A	A		-	A	
	close()	A		-	A	
open(F)		A		-	A	
write(B)		B		-	A	
	open(F)	B		-	A	Local processes
	read() → B	B		-	A	see writes immediately
	close()	B		-	A	
		B	open(F)	A	A	Remote processes
		B	read() → A	A	A	do not see writes...
		B	close()	A	A	
close()		B		A	B	... until close()
		B	open(F)	B	B	has taken place
		B	read() → B	B	B	
		B	close()	B	B	
		B	open(F)	B	B	
open(F)		B		B	B	
write(D)		D		B	B	
		D	write(C)	C	B	
		D	close()	C	C	
close()		D		C	D	
		D	open(F)	D	D	Unfortunately for P_3
		D	read() → D	D	D	the last writer wins
		D	close()	D	D	

4. Crash consistency

server崩溃的后果很严重，所有clients需要对本地file持怀疑态度，server也不再有clients及其持有文件的信息。解决方法：server恢复后给所有clients发送信息；client定期检查server

NFS vs. AFS

Workload	NFS	AFS	AFS/NFS
1. Small file, sequential read	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
2. Small file, sequential re-read	$N_s \cdot L_{mem}$	$N_s \cdot L_{mem}$	1
3. Medium file, sequential read	$N_m \cdot L_{net}$	$N_m \cdot L_{net}$	1
4. Medium file, sequential re-read	$N_m \cdot L_{mem}$	$N_m \cdot L_{mem}$	1
5. Large file, sequential read	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
6. Large file, sequential re-read	$N_L \cdot L_{net}$	$N_L \cdot L_{disk}$	$\frac{L_{disk}}{L_{net}}$
7. Large file, single read	L_{net}	$N_L \cdot L_{net}$	N_L
8. Small file, sequential write	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
9. Large file, sequential write	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
10. Large file, sequential overwrite	$N_L \cdot L_{net}$	$2 \cdot N_L \cdot L_{net}$	2
11. Large file, single write	L_{net}	$2 \cdot N_L \cdot L_{net}$	$2 \cdot N_L$

DAX文件系统

也称为持久内存感知文件系统，官方给出的解释是，既可以利用底层的块驱动程序将PM视作块存储，也可以将PM视作可字节寻址的存储。一般指的后者，现在支持DAX的文件系统有：EXT2,EXT4,XFS。支持DAX意味不再使用page cache，具体实现细节不明。

文件系统空闲空间管理(和file mapping并不是分开的?

1. Bitmap
2. list
3. Extent tree

B+树索引和哈希索引只是用来加速文件查询的？？

Extent tree通常是一棵基于B+树结构的树形索引，它将所有的数据块按照文件的ID号进行索引，并且使用Extent的起始块号和长度来描述一个Extent的范围。当需要读取或写入一个文件时，文件系统可以快速地根据Extent tree的索引信息找到该文件的Extent，然后读取或写入对应的数据块。

Extent tree是一种用于管理文件系统中文件数据块分配的数据结构。在文件系统中，每个文件都由一系列数据块组成，这些数据块可能在磁盘上不连续，需要一个数据结构来管理它们。Extent tree通过维护文件中每个连续数据块的起始位置和长度，来管理文件的数据块分配情况。

具体来说，Extent tree将文件的数据块按照顺序组织为若干个连续的Extent，每个Extent包含了一段连续的数据块。Extent tree将这些Extent以树的形式组织起来，每个节点代表一个Extent，包含了该Extent的起始位置和长度，以及指向子节点的指针。

当文件需要新增数据块时，Extent tree会搜索空闲的Extent，找到一个连续的Extent，并分配其中的一段数据块来存储新的数据。

文件系统映射

文件映射是将文件中的逻辑偏移量映射到底层设备上的物理位置。

文件通常由一系列逻辑块组成，逻辑块是文件系统中对文件进行逻辑划分的基本单位。每个逻辑块通常由固定数量的字节组成，这个数量通常是文件系统的块大小，一般为4KB或8KB。

- Hash

在查找文件或目录时非常快速，不需要遍历整个目录树。

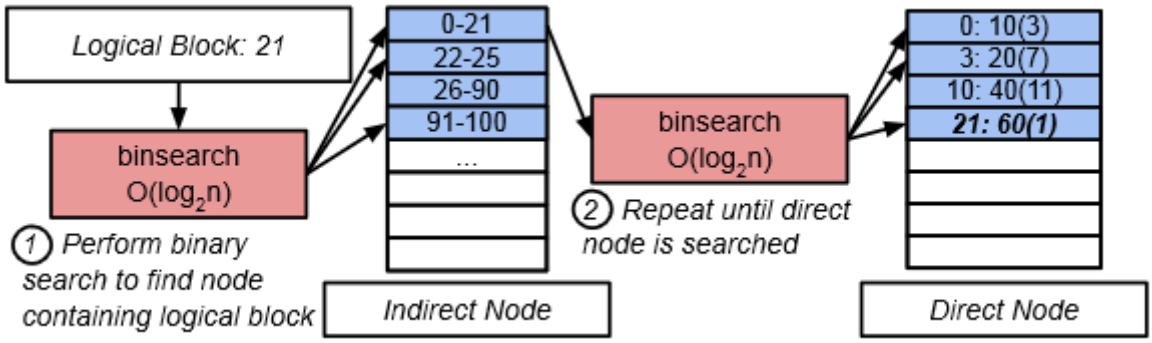
由于哈希函数的特性，哈希索引不能支持范围查询和按照名称的顺序遍历等操作。

对于空闲空间的管理，基于哈希的文件系统一般使用一些专门的算法来分配和回收空间

- Extent tree

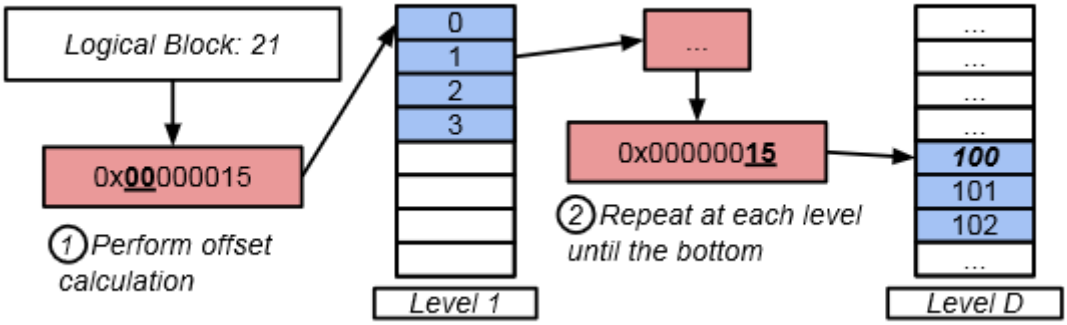
extent也有indirect，指向extent tree而不是数据块

一个查找案例：

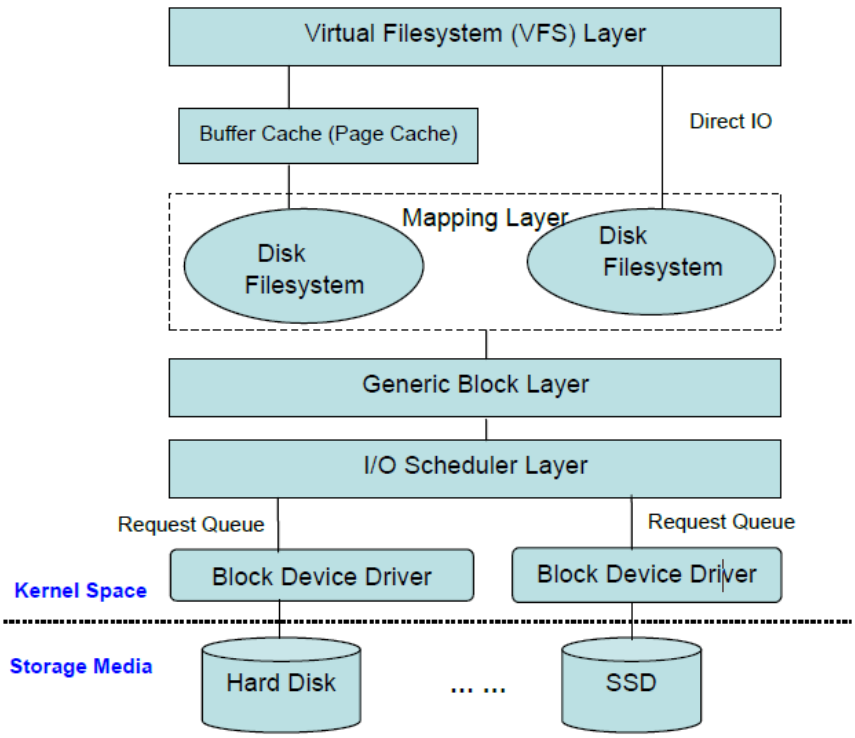


- Radix tree

按照块号的位数依次解析，基数树不如范围树紧凑，并且通常需要更多的间接性。



通用块设备层



映射层：内核需要确定要读取的数据在物理设备上的位置

通用块设备层：作为内核的一部分，粘合上层和底层，它知道块，扇区，段，页

Buffer cache和Page cache

Buffer(Buffer Cache)以块形式缓冲了块设备的操作，定时或手动的同步到硬盘，它是为了缓冲写操作然后一次性将很多改动写入硬盘，避免频繁写硬盘，提高写入效率。

Cache(Page Cache)以页面形式缓存了文件系统的文件，给需要使用的程序读取，它是为了给读操作提供缓冲，避免频繁读硬盘，提高读取效率。

> 不知道理解的对不对：

块设备概念

1. 扇区

块设备的最小寻址单元和数据传输单元，一般512字节

2. 块

VFS的数据传输单元，是扇区大小的整倍，但不超过页

3. 段（segments）

简单的DMA操作，只能传输磁盘上相邻的扇区。“分散/聚合”（scatter-gather）DMA操作，这种操作模式下，数据传输可以在多个非连续的内存区域中进行。一个段就是一个内存页面或一个页面的部分，它包含磁盘上相邻扇区的数据。这样一个“分散/聚合”DMA操作可能会涉及多个段。

页内磁盘布局：