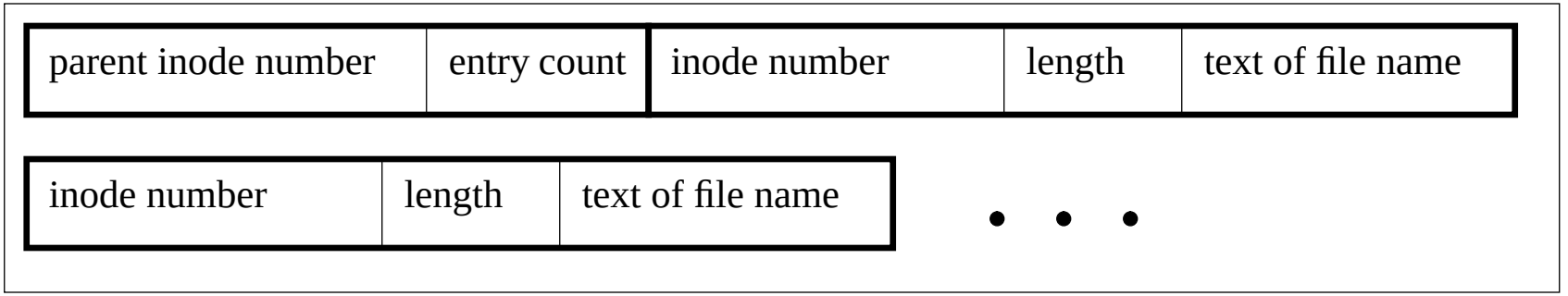
# 1.0 In-Inode Format

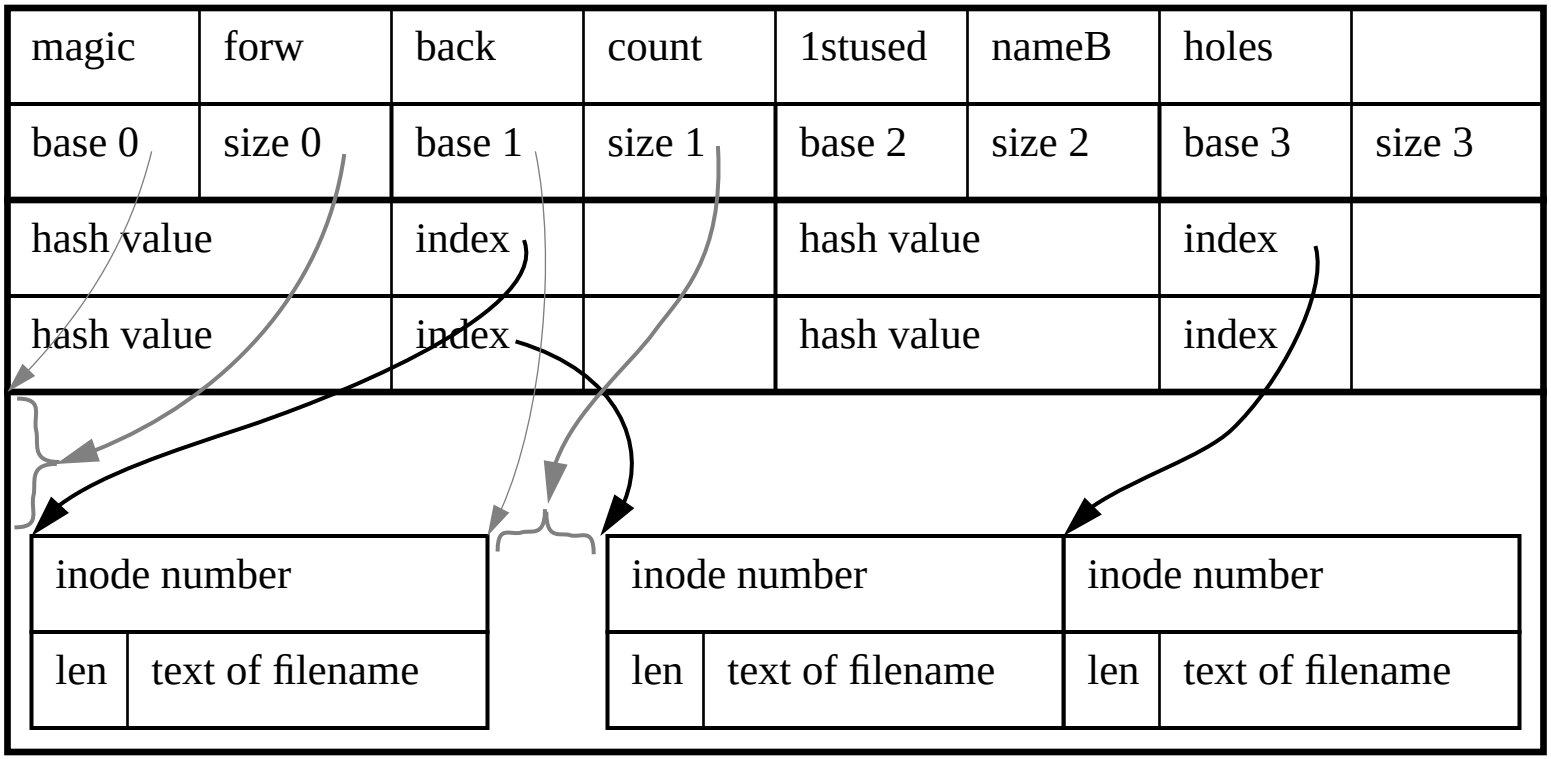
- **Directories stored inside an inode have this structure:**




- **Packed tight on byte boundaries.**
- **Inode size grows/shrinks to exactly required size.**
- **When it won't fit in the inode any more, transition to a single leaf node.**
- **No ordering of names, linear search.**
- **Parent inode number is explicit, self reference (eg: ".") is implicit.**

## 2.0 Leaf Blocks

- **Leaf nodes have this structure:**




- **Part of a doubly linked list at each level in the tree (forw, back)**
- **Manages free/used space in the block via limited-size, run-length coded, map (base*, size*)**
- **Tracks if there is free space not recorded in freemap (1stused, nameB, holes)**
- **Packed array of hash values gives offset within block of the rest of the information.**

## 3.0 Non-Leaf Nodes

- **Intermediate nodes have this structure:**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| magic | forw | back | count | freechn | freecnt | leafnext | |
| hash value | | blkno | | hash value | | blkno | |
| hash value | | blkno | | hash value | | blkno | |
| • • • | | | | | | | |

- **Part of a doubly linked list at each level in the tree (forw, back)**
- **Manages free/used space in the block via a simple count of packed entries (count)**
- **Manages free space in the "file" via a linked list of chunks (freechain, freecount) [not finished]**
- **Tracks if the next level down is a leaf node (leafnext)**
- **Array of hash values gives block number referencing all keys <= hashval.**

## 4.0 Working Data Structures

- **The arguments for a given operation:**

```
struct xfs_dir_name {
    char     *name;             /* string (maybe not NULL terminated) */
    int      namelen;           /* length of string (maybe no NULL) */
    uint     hashval;           /* hash value of name */
    xinum_t  inumber;           /* input/output inode number */
};

typedef long long xinum_t       /* inode number */
typedef unsigned short xblkno_t /* logical block number within directory */
```

## 4.0 Working Data Structures (continued)

- **Everything you'd want to know about a given block in the Btree:**

```
struct xfs_dir_state_blk {
    buf_t    *bp;              /* buffer containing block */
    xblkno   blkno;            /* blkno of buffer */
    int      index;            /* relevant index into block */
    uint     hashval;          /* last hash value in block */
    int      leafblk;          /* 1->blk is a leaf, 0->blk is a node */
};
```

- **The path back to the root that got you here:**

```
struct xfs_dir_state_path {
    int      active;           /* number of active levels */
    struct xfs_dir_state_blk blk[XFS_DIR_NODE_MAXDEPTH];
};
```

- **The complete working context for a given directory operation:**

```
struct xfs_dir_state {
    struct xfs_dir_name         *args;     /* filename arguments */
    trans_t                     trans;     /* transaction context */
    int                         inleaf;    /* insert into 1->lf, 0->splf */
    struct xfs_dir_state_path   path;      /* search/split paths */
    struct xfs_dir_state_path   altpath;   /* alternate path for join */
};
```

## 5.0 Leaf Add Algorithm

- **If there is space in the freemap, insert the entry:**

```
Find enough space in the freemap entry.
Force open space in the hashval array for the new entry.
Fill in a new entry in the hashval array.
Set up a name structure in the space from the freemap.
Copy the inode number, filename text, and text length into the name structure.
Update control info (firstused, namebytes, count, freemap, etc).
```

- **If we can't find room but there are "holes" in the block, compact the block:**

```
Kmem_alloc() a block sized chunk of RAM.
Make it look like an empty leaf node.
Copy all existing entries into the new leaf.
Copy the whole block back to the real leaf node.
```

- **If we compacted the block, try again looking for space in the freemap.**

```
Use the same algorithm as above.
```

- **Since the freemap is fixed size, the "holes" field tells us whether we have "lost" space from the freemap. "Holes" and the total of the map sizes tells us when to do garbage collection**
- **When allocating space from the freemap, preference is given to high addresses so as to reduce contention for the space immediately above the hashval array.**

## 6.0 Leaf Remove Algorithm

- **Look through freemap for relevant facts:**

```
If dying entry is adjacent to a freemap entry, remember this fact.
If map entry is adjacent to the end of the hashval table, add hashval to map.
```

- **If any adjacencies found with the dying entry, fix up the freemap:**

```
Including “before”, “after”, and “both” cases.
```

- **else, overwrite the smallest size entry in the map if it is smaller that our newly free space:**

```
Induces “holes” by dropping free space, requiring garbage collection later.
```

- **Update the control info for the leaf.**

```
Compress the dying entry out of the hashval array.
Fixup “firstused” if dying entry had the lowest address in the block.
Decrement “namebytes” to account for the dying filename.
```

- **Occasional garbage collection is better than complete management of the free space in a leaf.**

## 7.0 Btree Split Algorithm

- **Try to rebalance with an adjacent leaf before doing a split [future].**

```
Delays the work of splitting and increases storage efficiency.
```

- **Split the leaf node:**

```
Get the lowest numbered block off the “freechain” or grow the inode.
Fill in the new block as an empty leaf.
Balance the entries between the two blocks:
    Figure optimal balance of byte usage between blocks, given new name size.
    Compact destination block if it doesn’t have enough linear space.
    Move entries between blocks as needed to get into balance.
Link the new block into the linked list at this level.
Add the new entry to the correct leaf block (using “leaf add” algorithm).
```

- **Walk back up the path to the root, splitting nodes as required:**

```
If we don’t have to split the node, just add the new entry and return.
Get the lowest numbered block off the “freechain” or grow the inode.
Fill in the new block as an empty node.
Balance the entries between the two blocks:
    Simply total up the number of entries in both, half goes to each block.
    Force open a hole in the destination block if required.
Link the new block into the linked list at that level.
Insert the new entry into the correct node.
```

- **As we walk up the path, fixup changed hashvals to the root from each level:**

```
A new entry could change the last hashval in the block (which is known above).
```

- **If we split the root node, create a new root:**

```
Get the lowest numbered block off the “freechain” or grow the inode.
Copy the existing root into the new block.
Create an empty node at block 0.
Point to the relocated old root, and the node we just split from the old root.
```

## 8.0 Btree Join Algorithm

- **Join when leaf can be combined with a neighbor with 25% free space left in the block.**
- **Always keep lowered numbered blocks when joining any two blocks:**

```
This will shrink the directory over time.
```

- **Join the leaf node:**

```
Pick the lower numbered neighbor that will accommodate the names in this leaf.
Find the path from the root to our neighbor leaf.
Make the path from the dying block to the root our current path.
Unbalance the entries between the two blocks:
    If destination leaf has no holes, move all entries to destination leaf.
    If destination leaf has holes:
            Kmem_alloc() a buffer, make it look like an empty leaf, copy both
            leaves into the buffer, then copy the buffer back over the
            destination leaf.
Unlink the dying block from its neighbors on this level.
```

## 8.0 Btree Join Algorithm (continued)

- **Walk back up the current path to the root, joining nodes as required:**

```
Remove the reference to the dying block from this node.
If this node and a neighbor will fit with 25% to spare:
    Pick the lower numbered neighbor that will accommodate all the entries.
    Find the path from the root to our neighbor node.
    Make the path from the neighbor node to the root our current path.
Unbalance the entries between the two blocks:
    Force a hole in the destination node if required.
    Move all entries from the dying node to the destination node.
Unlink the new block from its neighbors at this level.
Insert the new entry into the correct node.
Fix hashvals in each block all the way to the root:
    A join could change the last hashval in the block (which is known above).
    Do this at every join so that we don't have to deal with multiple fixes.
```

- **If the root node only has one entry in it, delete the root:**

```
If the next level down is a leaf:
    Copy the leaf node to block 0.
    Reduce the inode size to one logical block.
If the next level down is not a leaf:
    Copy the freechain info from the old root to the only child of the root.
    Copy the only child of the root to block 0 (becoming the new root).
    Put the old child's block number onto the freelist.
```

## 9.0 Overall

- **Use multireader lock on the inode to protect against simultaneous modifications.**

```
Multiple readers in the directory at once, but still single threaded on the
root node of the Btree.
Only during lookups, blocks are released as soon as possible.
```

- **If freechain shows free blocks at the end of the inode, shrink inode [future].**

## 10.0 Attributes [all future]

- **Use same data structures and code as directory support.**

Too much in common to not make them the same.

- **Directory code and data structures will have to grow some to accommodate attribute req's.**

Will need some extra fields and semantics.

- **Small attribute values will be stored in leaf nodes adjacent to attribute name.**

Just add more stuff after the text of the name.

- **Large attribute values will be stored in dedicated block(s) that the leaf entry will point to.**

The entry in the leaf node will contain the starting block number of the value, and the length of the value in bytes.