Flexible See-Through Base VM Images

Akshay Sharma
Pranjal Singh

Course Project - Topics in Operating Systems

2024-25 - I Semester

Supervisor - Prof. Debadatta Mishra

1 Introduction

1.1 QCOW2 Disk Images - Copy-on-Write VM Clones

The QEMU emulator's default disk image format is QCOW2, or QEMU Copy on Write (version 2). As the name suggests, this format supports taking snapshots of disks for checkpointing or forking VMs. As the snapshot is modified over time, only the blocks (clusters of blocks, more formally) that are touched are copied with the changes made.

The base image is also known as the backing image.

1.2 See-through Base Images

A common use case for using backing images is to avoid storing redundant copies of common files. For example, system utilities (`/usr`), operating system images and ML datasets might be the same in multiple VMs in a cloud server. Further, these are not writable files.

For example, a cloud user may use a certain base VM as the default system, and modify files within it (for example, change the default shell from bash to csh).

However, system files receive periodic updates, including security fixes. At present, the user is required to make the update in each of the child snapshots. Over time, redundancy creeps in. We discuss the reasons for this in later sections.

2 Current Solutions and Limitations

Consider a simplified example where the disk block size is 4096 B and the snapshot cluster size is also 4096 B. As described earlier, our goal is to permit both the base image and the snapshot to be modified.

If possible, we also wish to fork *multiple children* from the base image.

2.1 Filesystem Consistency Requirements in EXT4

The EXT4 filesystem has the following entities corresponding to a file:

Entity	Location	Description	Size
Inode	Inode table	Create/update Inode	256 B
Inode bitmap	Dedicated inode bitmap blocks	Mark inode used	1 bit
Data blocks	Non-metadata blocks	Write data for non-empty files	FS block
Data block bitmap	Dedicated block bitmap	Mark block used	1 bit
Extent tree	Non-metadata blocks	Allocated/used as data blocks, block bitmap is set	FS block
Directory entry in parent	Parent's data blocks	Parent inode's extent tree and file length depend on children	<263 B
Superblock (global)	Beginning of disk, some backup copies	Free block/inode count, aggregate statistics	FS block(s)
Block Group Descriptor	Beginning of disk	Has free block count, free inode count	64 B

Thus, a file operation in EXT4 might need to update 9 different entities. This count excludes the block bitmap bits for the extent tree, and if the parent directory's data blocks are full, then its next data block, modified extent tree (which may need even more blocks), corresponding block bitmap bits, the parent inode's extent status and parent inode's file length entry.

In brief, filesystem consistency is a global notion. It is not easy to isolate some part of a filesystem (or an update) and declare it consistent/safe.

Concrete examples of consistency issues are discussed in the next section.

2.2 Consistency Issues in EXT4

Consider the following sequences:

- Data block 1050 is free when snapshot is taken
- Child FS writes a new file `/snapshot_1050` to **block 1050 of the snapshot** and CoW mapping of block 1050 to base image is overwritten with the new data.
- Base image (being operated by parent VM) is unaware of above write. Parent creates file `/base_1050` which is written to **block 1050 of the base image**.
- Now, it is possible that the child VM (running the disk snapshot) sees the directory entry and inode for `/base_1050` correctly.
- Child reads `/base_1050`, but the disk access is redirected to the new data block containing `/snapshot_1050`.

or,

- Assuming inodes have size 256 B, **16 inodes fit in a disk block**.
- Inodes 5 and 6 (residing in first block of inode table) were free when the snapshot was taken.
- Child VM uses **inode 5 in block 0** for a new file `/snapshot_5` and replaces the CoW mapping by a new block. Inodes 0 to 15 reside in the first block.
- Parent uses **inode 6, also in block 0** for a new file `/base_6` in the base image.
- Child sees the directory entry for `/base_6` and reads inode 6
- The read is redirected to the new block and the child reads garbage values

More generally, the base VM might use a **(a) disk block that the child had earlier dirtied** and has a separate copy of (data blocks and extent trees have 4 KB granularity). Or, the base VM might use some **(b) part of a disk block, of which the child has dirtied a different part**, and created a copy (for remaining metadata entities with smaller size).

2.3 Overview of Limitations

In a nutshell, a VM modifying a disk snapshot needs to take responsibility for the 9 filesystem entities in the snapshot (listed above), as well as the *same 9 entities in the base image.* Block layer writes are done at a granularity of 512 B or 4096 B. Moreover, QCOW2 snapshots have a default cluster size of 64 KB.

Filesystem consistency is guaranteed if (i) the base image is modified , but the child VM abstains from writing, or (ii) the base image is not modified (and the child VM has virtual “control” over the base image), but the Copy-on-Write snapshot is dirtied.

This also holds when multiple copies of a snapshot are created and used by multiple VMs. (Pitfall: we don’t mean modifying a live disk here. We mean modifying the base image after taking the snapshot, regardless of the child image being live/unmounted.)

These complications arise because the disk image is being CoW-ed at the granularity of 4 KB (64 KB by default). At the VFS layer, for instance, we would only need to worry about clashes in the names of files.

2.4 Alternate Solutions: Partitioning the Disk

Disk partitions can be kept consistent independently, unlike a directory or a file (with the 9 entities listed earlier). One can partition a disk and assign a partition to a single writer.

Under this scheme, it is also easy for k sibling VMs to have a writable partition, and read the partitions assigned to the remaining $k-1$ siblings. FS consistency is maintained because a partition has a single writer. However, the disk allocation size under this scheme is inflexible. One workaround could be to have a large number of partitions and dynamically change ownership.

2.5 Alternate Solutions: Network Filesystem

NFS mounts spare the VM filesystem the responsibility of filesystem consistency because sharing is done at the VFS view. That would also permit simultaneous, direct modifications to the base image's and snapshot's designated directories. Additionally, space savings would be even larger if devices across multiple nodes can share data. However, NFS mounts suffer from a large access latency and low bandwidth.

3 Disallowing Writes to Selected Block Groups

In this section, we discuss our current approach.

3.1 EXT4 Block Groups

To avoid frequent disk seeks, EXT4 divides the disk into block groups (BG), which are physically contiguous portions of the disk. To the extent possible, all blocks in a file as well as its metadata elements are stored within a block group.

The block bitmap is *always* stored within the containing block group. An inode bitmap and inode table are also neighbours on disk, although the corresponding file's blocks may be in a different block group.

EXT4 also has an extension to block groups, known as flexible block groups which is functionally similar to block groups. It links multiple block groups into a virtual BG (typically 16) and stores their metadata For simplicity, we disable this feature, although we can accomodate it easily owing to physical contiguity. To disable it (when creating the partition):

```
# mkfs.ext4 -O ^flex_bg [options] device
```

Thus, the layout of a typical BG in a partition without flexible BGs is:
(`bash# echo stats | debugfs /dev/your_ext4_disk`)

- Block bitmap at X
- Inode bitmap at X+1
- Inode table at X+2 (typically 8K Inode blocks spanning 8Kx256 bytes, 512 disk blocks)
- Data blocks from X+2+512 to X + 2^{15}

With flexible BGs, (say 16 BGs per virtual BG) it is

- 16 block bitmaps: X to X + 15
- 16 Inode bitmaps: X + 16 to X + 31
- Inode table from X + 32 to Y
- Data blocks from Y+1 to X + (16 $\times 2^{15}$) - 1

This permits storing large files contiguously on disk.

3.2 Restricting Writes to Designated Block Groups

By assigning a block group to a single writer, FS consistency can be maintained even if the partition as a whole is operated upon by multiple entities. Some disparities crop up in the free block/inode count, which can be fixed by `fsck`. Our approach is to writes to designated BGs in the base image as well as the snapshot.

The semantics of this assignment are:

Ownership	Operation	Permitted	Notes
Owned	Read	Y	Allowed
Owned	Write	Y	New blocks & extents must be in owned region
Owned	Creation	Y	New Inode must be in owned region. Parent's inode, extents and data must be in owned region
Owned	Deletion	Y	Inode must be in owned region. Parent's inode, extents and data must be in owned region
Not Owned	Read	Y	Don't write access timestamp
Not Owned	Write, Create, Update	N	Not allowed
Superblock (shared)	Mount	Y	Might need to maintain a separate copy for parent and child
Superblock (shared)	Update	Parent Only	Child can have a separate copy
BG descriptors	Update	Y	Up-to-date BG descriptors are not absolutely necessary, but better to let the child maintain a separate copy

Additionally, one directory needs to be created in the BGs assigned to the child, whose parent is in a BG assigned to the base image's writer. These blocks need to be kept fixed through the disk image lifetime. This is a special operation, as it is not permitted under the semantics described above. This can be done when the disk is being created.

4 Results and Observations

4.1 Partitioning the Disk Image

We created a secondary disk for VMs and took two snapshots of the same, corresponding to two child VMs. Two EXT4 partitions were created on the disk, of which the first was designated to be updated only in the base image, and the second only in the snapshots.

QEMU locks QCOW2 files upon exposing them as disks to VMs. We modified QEMU to not do so.

Linux tries storing disk metadata and data in memory to avoid disk IO for performance, in the page cache. (See `$ top`'s output.) The page cache can be flushed using this command:

```
(as root)# echo 3 > /proc/sys/vm/drop_caches
```

From the behaviour of this command, it appears that directory metadata is not cleared. (Some parts corresponding to open files cannot be cleared.) We used `-o sync` in mounting these partitions, and observed that file contents are read and cached upon the first read usually. Regular file contents are always observed as updated by the writer after dropping the cache.

4.2 Disallowing Allocations to Selected Block Groups

We attempted to assign block group 1 to the snapshot and the remaining block groups to the base image for modifications.

For simplicity, we used a disk partition without a journal and copies of the superblock. (`mkfs.ext4 -O ^has_journal,sparse_super2 -E num_backup_sb=0`. See `mke2fs.conf(5)`).

4.2.1 Blocking Writes at the QEMU layer

Typically, if a write fails, the filesystem attempts to write the same data to a different sector. Relying on this filesystem feature, we blocked writes to the designated block groups at the QEMU block layer.

This did not work, as the files did not persist through a remount. Lengthy error messages are printed to the console.

4.2.2 Filling the Block Bitmap

Our first attempt was to present a modified view to the filesystem in which no blocks were available for writes. This can be done in the function `ext4_read_block_bitmap_nowait()`. Similar changes need to be made in the inode bitmap. This causes the (solvable) issue that upon unmount, the all-ones bitmap is flushed back to disk from the page cache. We did not test this extensively as we moved on to modifying EXT4's block allocation (next section).

4.2.3 Modifying Block and Inode Allocation Helpers

Determining a file's ownership: As mentioned earlier, a directory needs to be reserved for the child's data. We assume this is done before the child's BGs are dirtied, ie when creating the partition. Under this assumption, the residency of an inode (within the family of owned BGs or not) is the same as that of its contents. Thus, checking either the inode or any data block of a file is sufficient to determine the effective owner. We made the following changes:

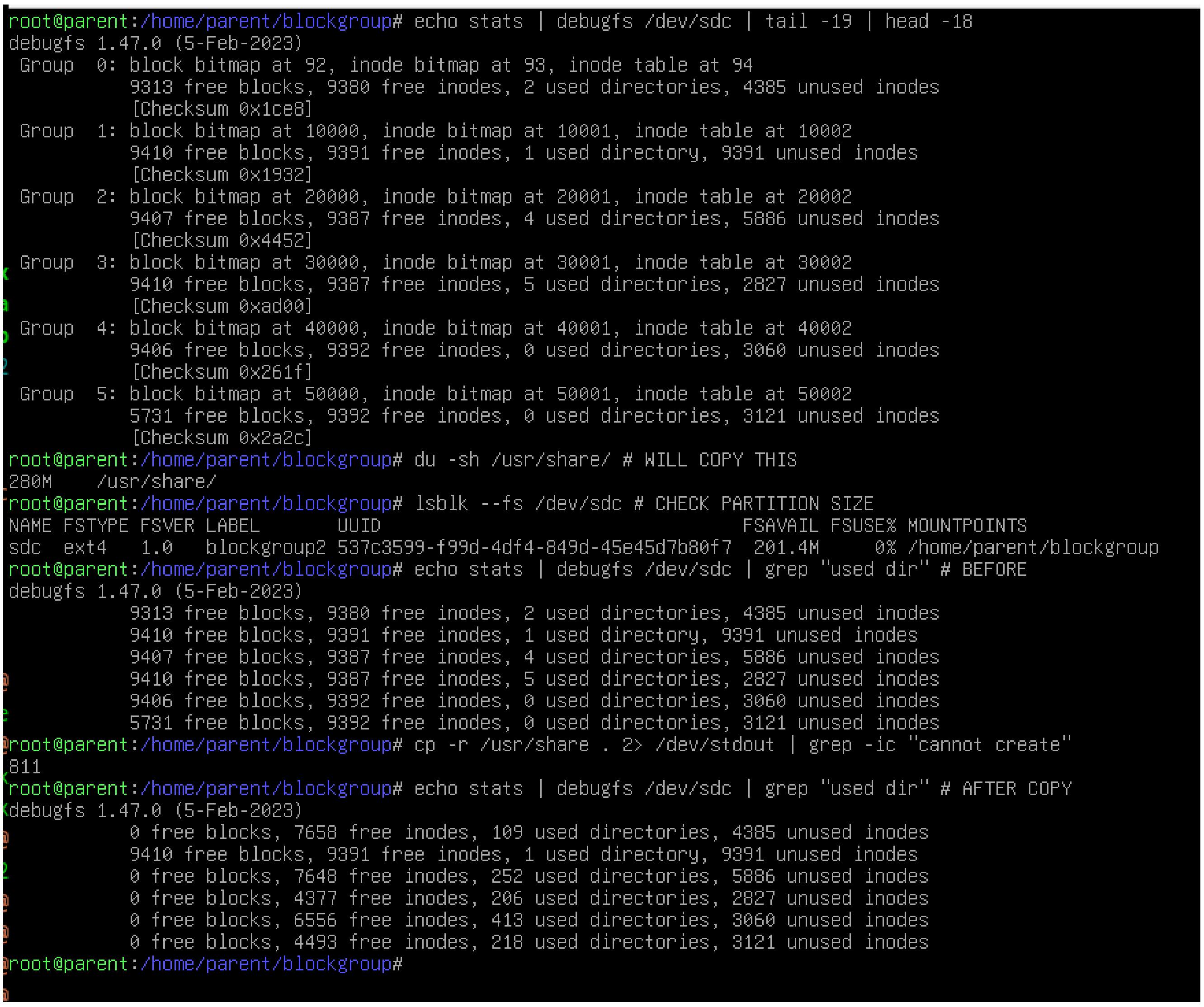
- `ext4_mb_regular_allocator`, the multi-block allocator function calls a helper, `ext4_mb_good_group_nolock` to determine if a block group has sufficient free space. We modified this function to ignore non-owned block groups.
- The function `__ext4_new_inode` allocates inodes from the pool of unused inodes. We modified it to ignore non-owned block groups.
- To disallow writes to non-owned files/directories, we modified `ext4_file_open`. For files opened in write mode, the inode number is checked to determine ownership and allow/stop writes. The inode of the parent directory is also checked, as files cannont be created. This function is EXT4's interface to the VFS and is called even if the file and its metadata are in memory.
- At the QEMU block layer, writes to the corresponding BGs can be disallowed. However, one would need to account for the superblock separately. We were able to isolate writes without enabling such filtering.
- To ease debugging with `gdb`, we made EXT4 a part of the kernel image instead of a loadable module (and incurred higher build times).

This approach succeeds in stopping illegal writes and allocating blocks only from permitted BGs. However, we were not able to stop illegal file creation. File creation happens before `ext4_file_open()` is called, and needs to be dealt with separately.

Figures 1, 2 and 3 verify our changes to the allocator.

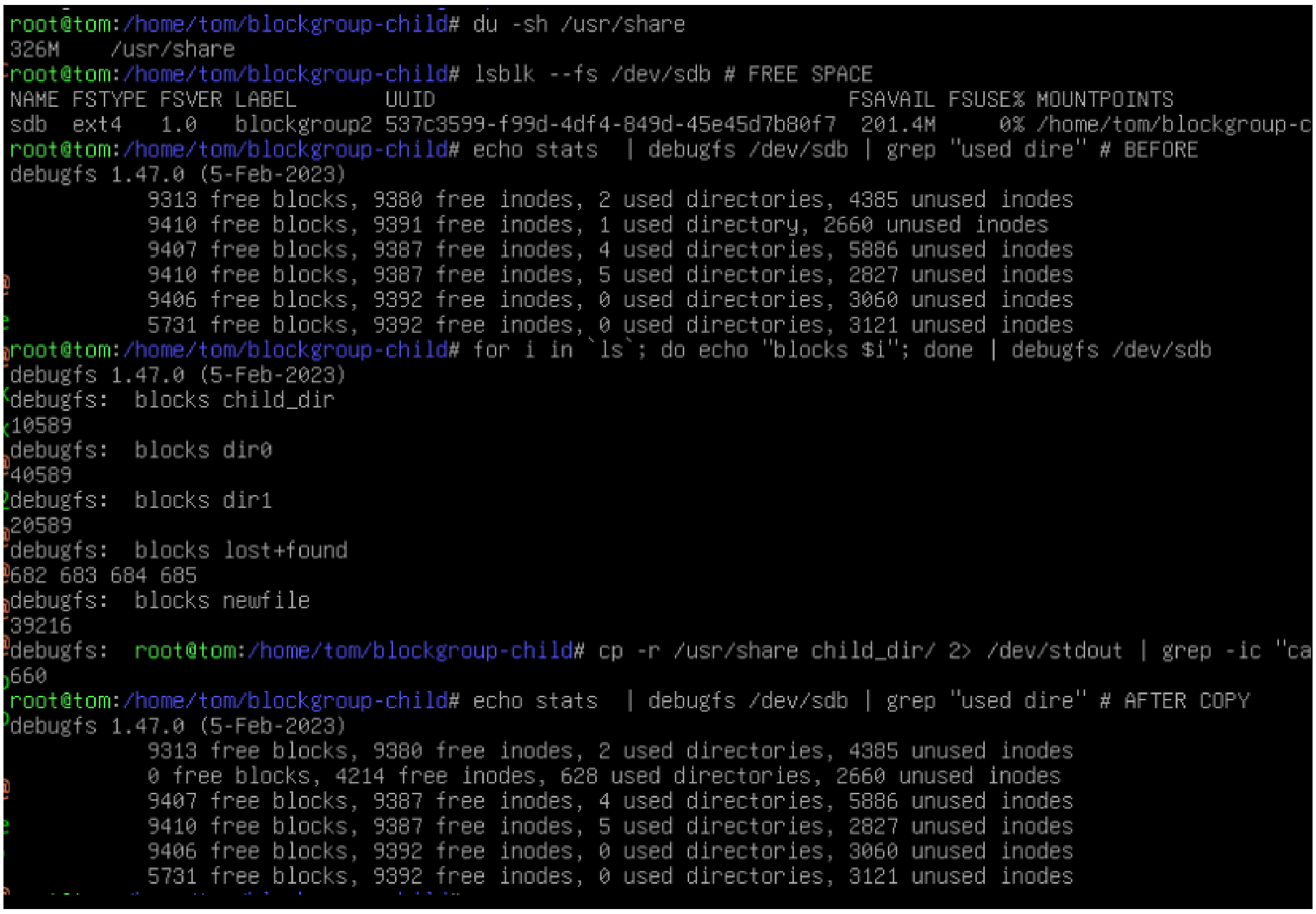
4.2.4 Concurrency and Page Cache

Under the above approach, directory contents and file contents are usually loaded only on the first access. Updates in the parent image done after the first read are not visible immediately.



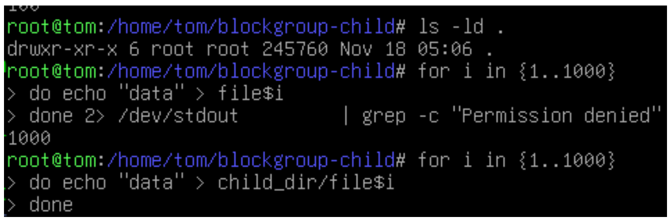
```
root@parent:/home/parent/blockgroup# echo stats | debugfs /dev/sdc | tail -19 | head -18
debugfs 1.47.0 (5-Feb-2023)
 Group  0: block bitmap at 92, inode bitmap at 93, inode table at 94
           9313 free blocks, 9380 free inodes, 2 used directories, 4385 unused inodes
           [Checksum 0x1ce8]
 Group  1: block bitmap at 10000, inode bitmap at 10001, inode table at 10002
           9410 free blocks, 9391 free inodes, 1 used directory, 9391 unused inodes
           [Checksum 0x1932]
 Group  2: block bitmap at 20000, inode bitmap at 20001, inode table at 20002
           9407 free blocks, 9387 free inodes, 4 used directories, 5886 unused inodes
           [Checksum 0x4452]
 Group  3: block bitmap at 30000, inode bitmap at 30001, inode table at 30002
           9410 free blocks, 9387 free inodes, 5 used directories, 2827 unused inodes
           [Checksum 0xad00]
 Group  4: block bitmap at 40000, inode bitmap at 40001, inode table at 40002
           9406 free blocks, 9392 free inodes, 0 used directories, 3060 unused inodes
           [Checksum 0x261f]
 Group  5: block bitmap at 50000, inode bitmap at 50001, inode table at 50002
           5731 free blocks, 9392 free inodes, 0 used directories, 3121 unused inodes
           [Checksum 0x2a2c]
root@parent:/home/parent/blockgroup# du -sh /usr/share/ # WILL COPY THIS
280M    /usr/share/
root@parent:/home/parent/blockgroup# lsblk --fs /dev/sdc # CHECK PARTITION SIZE
NAME FSTYPE FSVER LABEL       UUID                                 FSAVAIL FSUSE% MOUNTPOINTS
sdc  ext4   1.0   blockgroup2 537c3599-f99d-4df4-849d-45e45d7b80f7  201.4M     0% /home/parent/blockgroup
root@parent:/home/parent/blockgroup# echo stats | debugfs /dev/sdc | grep "used dir" # BEFORE
debugfs 1.47.0 (5-Feb-2023)
           9313 free blocks, 9380 free inodes, 2 used directories, 4385 unused inodes
           9410 free blocks, 9391 free inodes, 1 used directory, 9391 unused inodes
           9407 free blocks, 9387 free inodes, 4 used directories, 5886 unused inodes
           9410 free blocks, 9387 free inodes, 5 used directories, 2827 unused inodes
           9406 free blocks, 9392 free inodes, 0 used directories, 3060 unused inodes
           5731 free blocks, 9392 free inodes, 0 used directories, 3121 unused inodes
root@parent:/home/parent/blockgroup# cp -r /usr/share . 2> /dev/stdout | grep -ic "cannot create"
811
root@parent:/home/parent/blockgroup# echo stats | debugfs /dev/sdc | grep "used dir" # AFTER COPY
debugfs 1.47.0 (5-Feb-2023)
           0 free blocks, 7658 free inodes, 109 used directories, 4385 unused inodes
           9410 free blocks, 9391 free inodes, 1 used directory, 9391 unused inodes
           0 free blocks, 7648 free inodes, 252 used directories, 5886 unused inodes
           0 free blocks, 4377 free inodes, 206 used directories, 2827 unused inodes
           0 free blocks, 6556 free inodes, 413 used directories, 3060 unused inodes
           0 free blocks, 4493 free inodes, 218 used directories, 3121 unused inodes
root@parent:/home/parent/blockgroup#
```

Figure 1: Base Image: Second block group is not used even if others are full



```
root@tom:/home/tom/blockgroup-child# du -sh /usr/share
326M    /usr/share
root@tom:/home/tom/blockgroup-child# lsblk --fs /dev/sdb # FREE SPACE
NAME FSTYPE FSVER LABEL       UUID                                 FSAVAIL FSUSE% MOUNTPOINTS
sdb  ext4   1.0   blockgroup2 537c3599-f99d-4df4-849d-45e45d7b80f7  201.4M     0% /home/tom/blockgroup-c
root@tom:/home/tom/blockgroup-child# echo stats  | debugfs /dev/sdb | grep "used dire" # BEFORE
debugfs 1.47.0 (5-Feb-2023)
           9313 free blocks, 9380 free inodes, 2 used directories, 4385 unused inodes
           9410 free blocks, 9391 free inodes, 1 used directory, 2660 unused inodes
           9407 free blocks, 9387 free inodes, 4 used directories, 5886 unused inodes
           9410 free blocks, 9387 free inodes, 5 used directories, 2827 unused inodes
           9406 free blocks, 9392 free inodes, 0 used directories, 3060 unused inodes
           5731 free blocks, 9392 free inodes, 0 used directories, 3121 unused inodes
root@tom:/home/tom/blockgroup-child# for i in `ls`; do echo "blocks $i"; done | debugfs /dev/sdb
debugfs 1.47.0 (5-Feb-2023)
debugfs:  blocks child_dir
10589
debugfs:  blocks dir0
40589
debugfs:  blocks dir1
20589
debugfs:  blocks lost+found
682 683 684 685
debugfs:  blocks newfile
39216
debugfs:  root@tom:/home/tom/blockgroup-child# cp -r /usr/share child_dir/ 2> /dev/stdout | grep -ic "ca
660
root@tom:/home/tom/blockgroup-child# echo stats  | debugfs /dev/sdb | grep "used dire" # AFTER COPY
debugfs 1.47.0 (5-Feb-2023)
           9313 free blocks, 9380 free inodes, 2 used directories, 4385 unused inodes
           0 free blocks, 4214 free inodes, 628 used directories, 2660 unused inodes
           9407 free blocks, 9387 free inodes, 4 used directories, 5886 unused inodes
           9410 free blocks, 9387 free inodes, 5 used directories, 2827 unused inodes
           9406 free blocks, 9392 free inodes, 0 used directories, 3060 unused inodes
           5731 free blocks, 9392 free inodes, 0 used directories, 3121 unused inodes
```

Figure 2: Snapshot/child Image: Blocks and inodes are allocated from the second BG only.



```
root@tom:/home/tom/blockgroup-child# ls -ld .
drwxr-xr-x 6 root root 245760 Nov 18 05:06 .
root@tom:/home/tom/blockgroup-child# for i in {1..1000}
> do echo "data" > file$i
> done 2> /dev/stdout         | grep -c "Permission denied"
1000
root@tom:/home/tom/blockgroup-child# for i in {1..1000}
> do echo "data" > child_dir/file$i
> done
```

Figure 3: Illegal writes in the snapshot are stopped and legal writes are allowed

Empirically, clearing the page cache (described earlier) refreshes file contents but not directory entries. A complete remount is needed to refresh directory entries. Thus, this arrangement would not perform if data is being shared dynamically.

4.2.5 Possible Extensions

- The base image can use the child-owned regions as well, if it maintains similar isolation.
- Rather than reserving certain block groups, the child's view can be modified to include some block groups not present in the parent. This can save space and simplify the design.
- If one binds certain directories to a BG(s) such that all directory contents wil be within those BGs, that increases the granularity of sharing to a directory (from a block) and can allow the child and parent to have separate contents/views of a directory.