Fault Isolation and Quick Recovery in Isolation File Systems

Lanyue Lu
Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

University of Wisconsin - Madison

File-System Availability Is Critical

File-System Availability Is Critical

- Main data access interface
  - ➡ desktop, laptop, mobile devices, file servers

File-System Availability Is Critical

Main data access interface

- desktop, laptop, mobile devices, file servers

A wide range of failures

- resource allocation, metadata corruption
- failed I/O operations, incorrect system states

File-System Availability Is Critical

Main data access interface

- desktop, laptop, mobile devices, file servers

A wide range of failures

- resource allocation, metadata corruption
- failed I/O operations, incorrect system states

A small fault can cause global failures

- e.g., a single bit can impact the whole file system

File-System Availability Is Critical

Main data access interface

- desktop, laptop, mobile devices, file servers

A wide range of failures

- resource allocation, metadata corruption
- failed I/O operations, incorrect system states

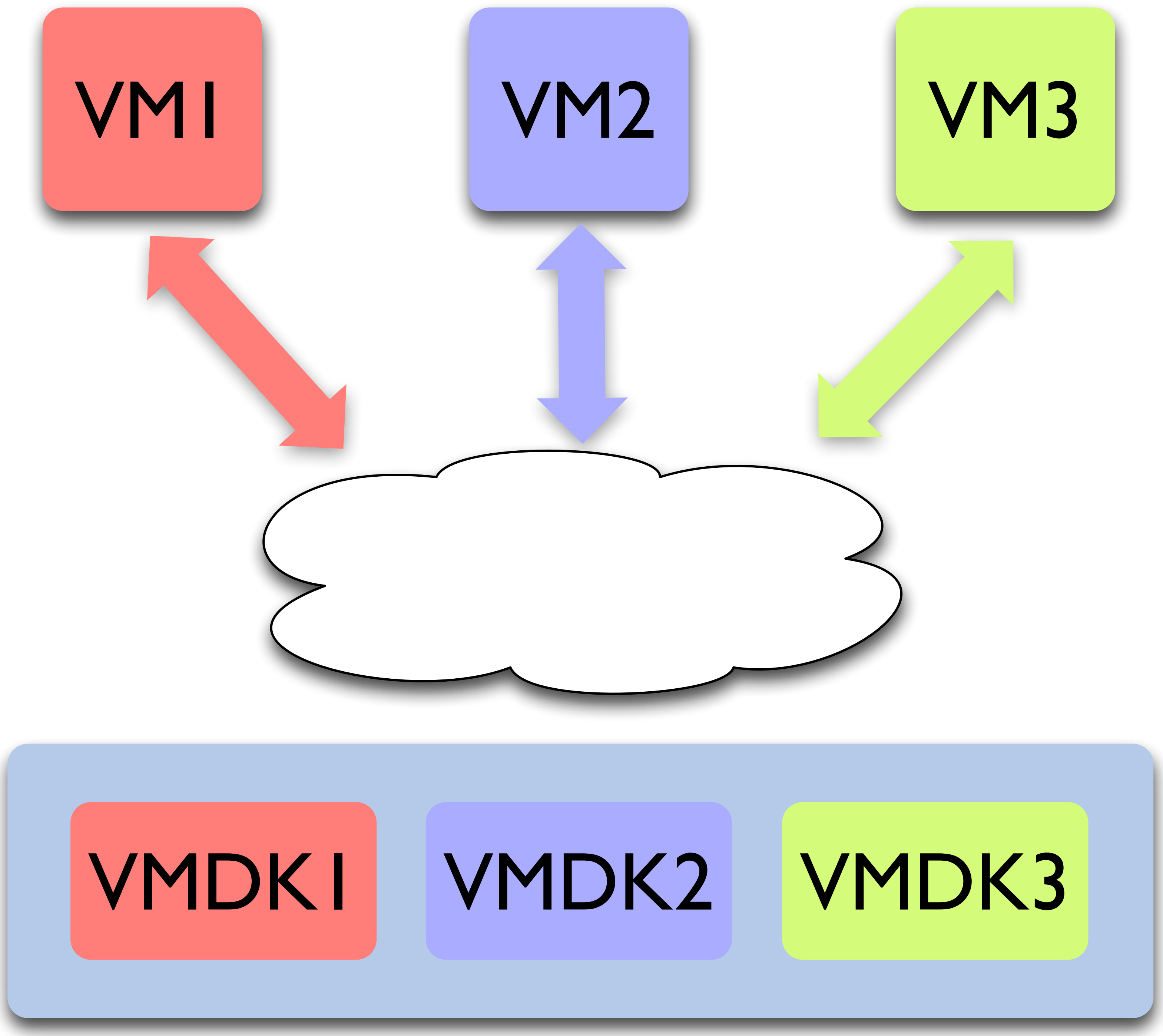
A small fault can cause global failures

- e.g., a single bit can impact the whole file system

Global failures considered harmful

- read-only, crash

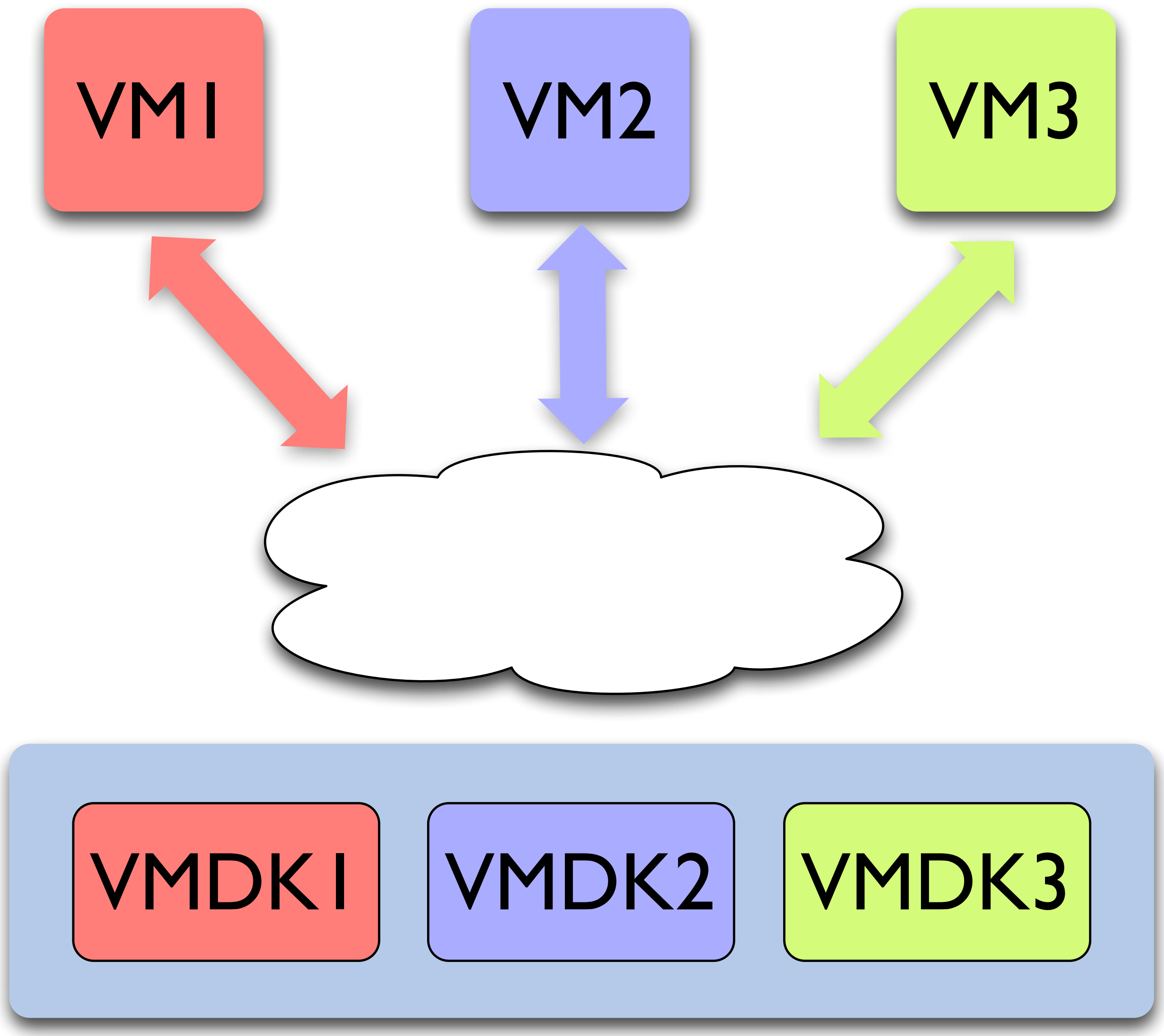
Server Virtualization

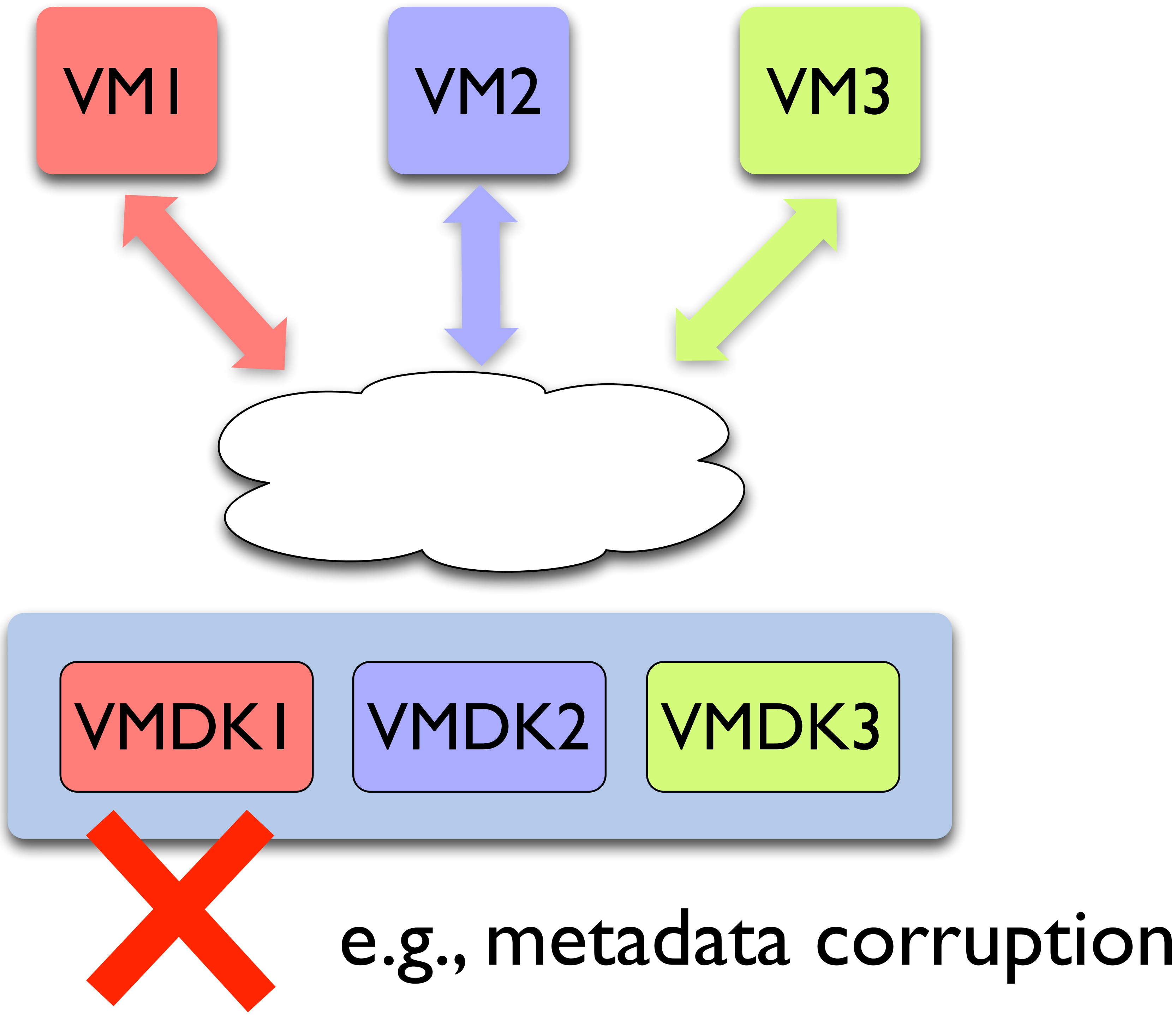


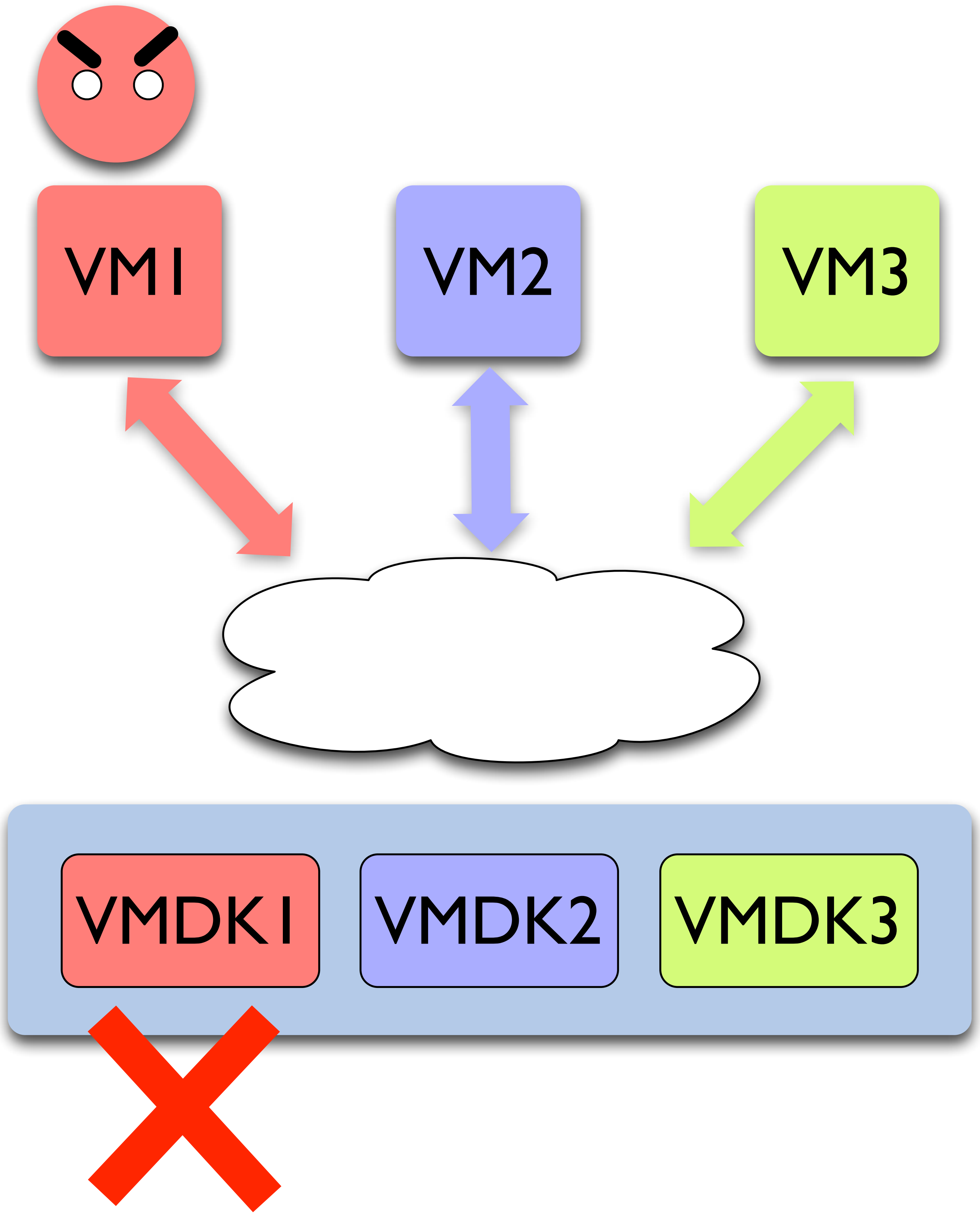
Guest virtual machines

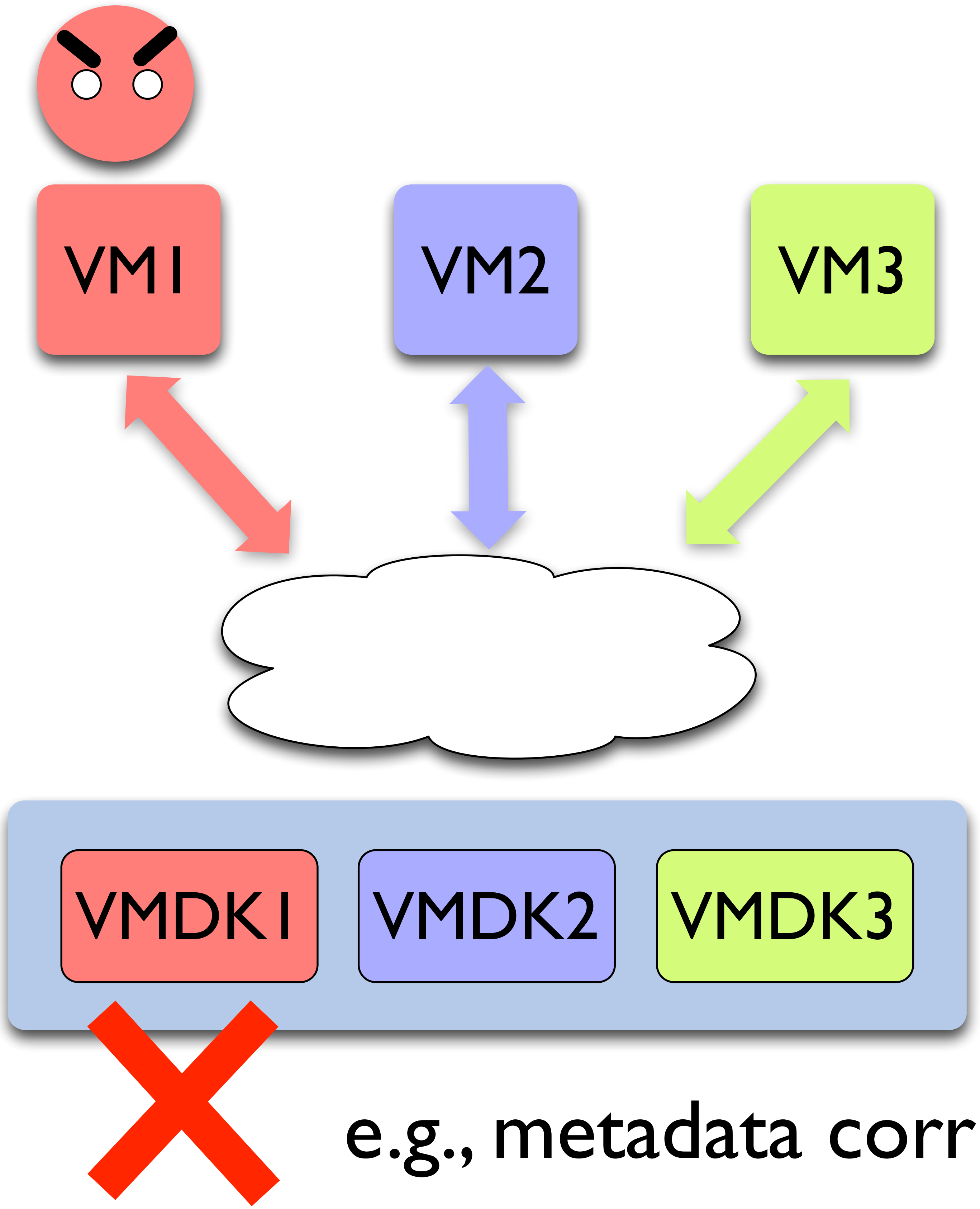
Hypervisor

Shared file system

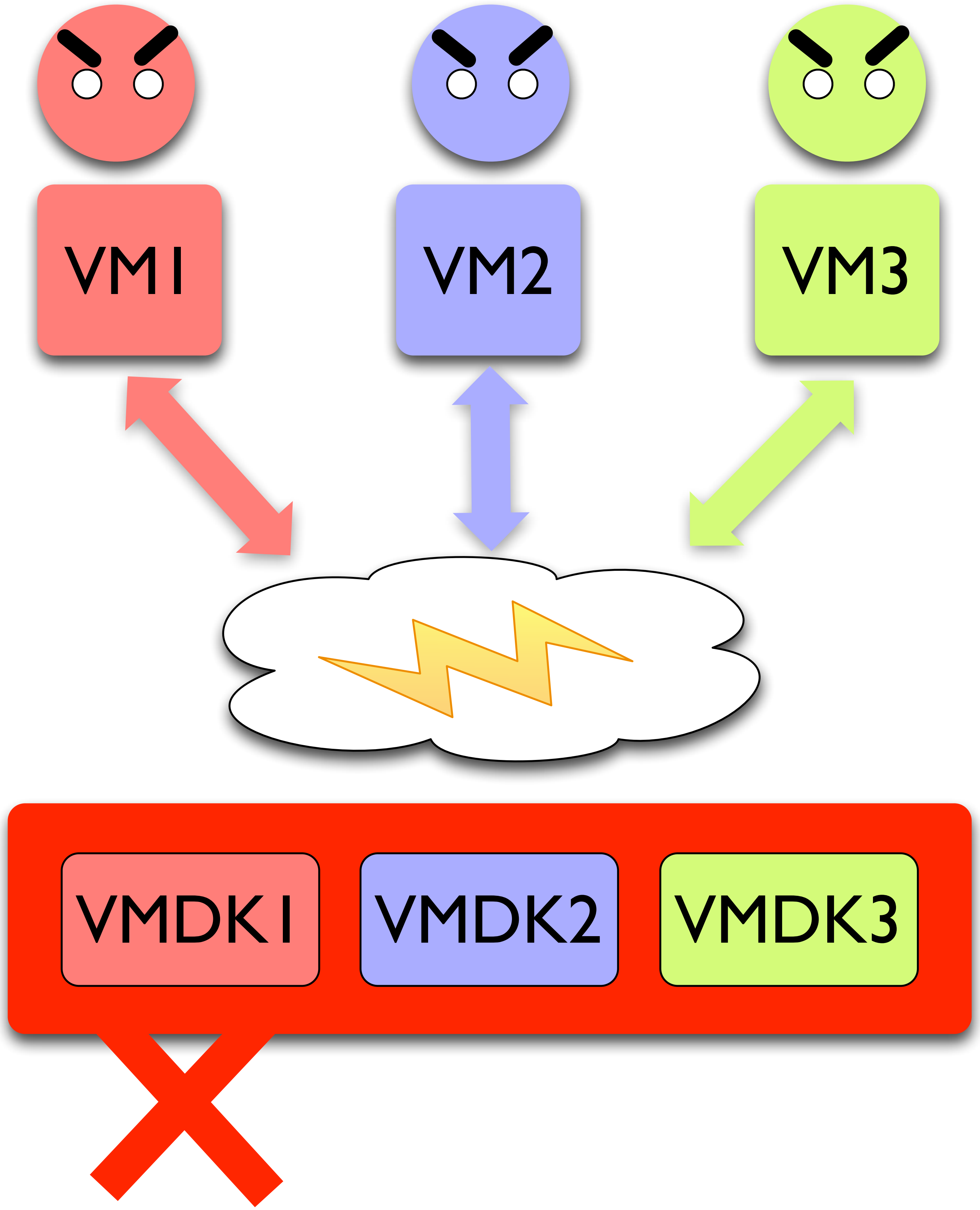

VM1
VM2
VM3
VMDK1
VMDK2
VMDK3


VM1
VM2
VM3
VMDK1
VMDK2
VMDK3
e.g., metadata corruption


VM1
VM2
VM3
VMDK1
VMDK2
VMDK3

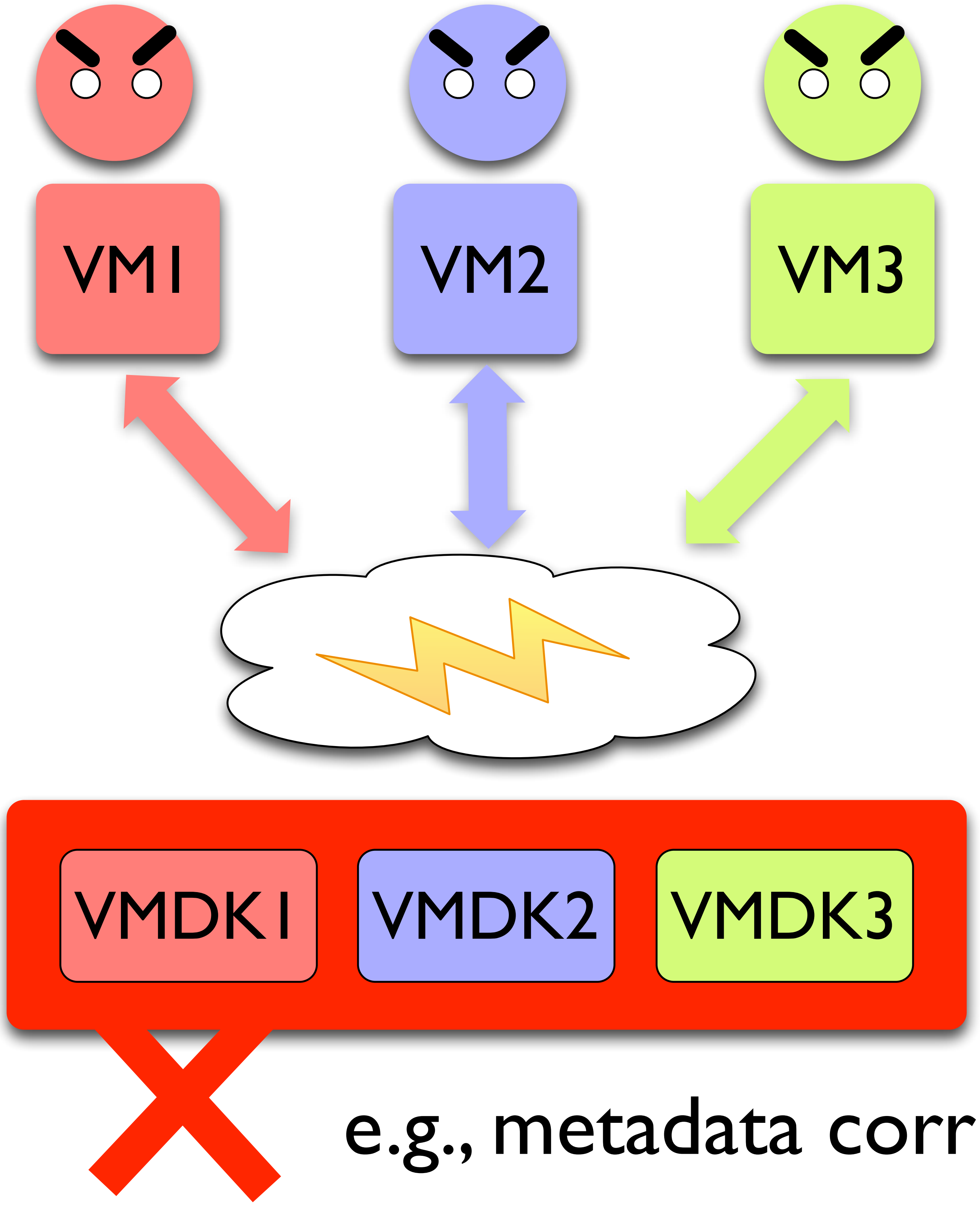


e.g., metadata corruption



All VMs
are affected

ReadOnly
or
Crash



All VMs
are affected

ReadOnly
or
Crash

e.g., metadata corruption

Our Solution

Our Solution

A new abstraction for fault isolation

- support multiple independent fault domains
- protect a group of files for a domain

Our Solution

A new abstraction for fault isolation

- support multiple independent fault domains
- protect a group of files for a domain

Isolation file systems

- fine-grained fault isolation
- quick recovery

Introduction

Study of Failure Policies

Isolation File Systems

Challenges

Questions to Answer

Questions to Answer

What global failure policies are used ?

- failure types
- number of each type

Questions to Answer

What global failure policies are used ?

- failure types
- number of each type

What are the root causes of global failures ?

- related data structures
- number of each cause

Methodology

Methodology

Three major file systems

- Ext3 (Linux 2.6.32), Ext4 (Linux 2.6.32)
- Btrfs (Linux 3.8)

Methodology

Three major file systems

- Ext3 (Linux 2.6.32), Ext4 (Linux 2.6.32)
- Btrfs (Linux 3.8)

Analyze source code

- identify types of global failures
- count related error handling functions
- correlate global failures to data structures

Q1:

What global failure policies

are used ?

Global Failure Policies

Global Failure Policies

Definition

- a failure which impacts all users of the file system or even the operating system

Global Failure Policies

Definition

- a failure which impacts all users of the file system or even the operating system

Read-Only

- e.g., ext3_error():
  - mark file system as read-only
  - abort the journal

Read-Only Example

```
ext3/balloc.c, 2.6.32

read_block_bitmap(...){

 1    bitmap_blk = desc->bg_block_bitmap;
 2    bh = sb_getblk(sb, bitmap_blk);
 3    if (!bh){
 4        ext3_error(sb, "Cannot read block
                          bitmap");
          return NULL;
      }
}
```

Read-Only Example

```
ext3/balloc.c, 2.6.32

read_block_bitmap(...){

    bitmap_blk = desc->bg_block_bitmap;
    bh = sb_getblk(sb, bitmap_blk);
    if (!bh){
        ext3_error(sb, “Cannot read block
                        bitmap”);
        return NULL;
    }
}
```

Read-Only Example

```
ext3/balloc.c, 2.6.32

read_block_bitmap(...){

 1    bitmap_blk = desc->bg_block_bitmap;
 2    bh = sb_getblk(sb, bitmap_blk);
 3    if (!bh){
 4        ext3_error(sb, "Cannot read block
                         bitmap");
          return NULL;
      }
}
```

Read-Only Example

```
ext3/balloc.c, 2.6.32

read_block_bitmap(...){

 1    bitmap_blk = desc->bg_block_bitmap;
 2    bh = sb_getblk(sb, bitmap_blk);
 3    if (!bh){
 4        ext3_error(sb, "Cannot read block
                              bitmap");
          return NULL;
      }
}
```

Read-Only Example

```
ext3/balloc.c, 2.6.32

read_block_bitmap(...){

    bitmap_blk = desc->bg_block_bitmap;
    bh = sb_getblk(sb, bitmap_blk);
    if (!bh){
        ext3_error(sb, “Cannot read block
                        bitmap”);
        return NULL;
    }
}
```

Global Failure Policies

Definition

- a failure which impacts users of the file system or even the operating system

Read-Only

- e.g., ext3_error():
  - mark file system as read-only
  - abort the journal

Crash

- e.g., BUG(), ASSERT(), panic()
- crash the file system or operating system

Crash Example

```
btrfs/disk-io.c, 3.8

open_ctree(...) {

    root->node = read_tree_block(...);
    BUG_ON(!root->node);
```

Crash Example

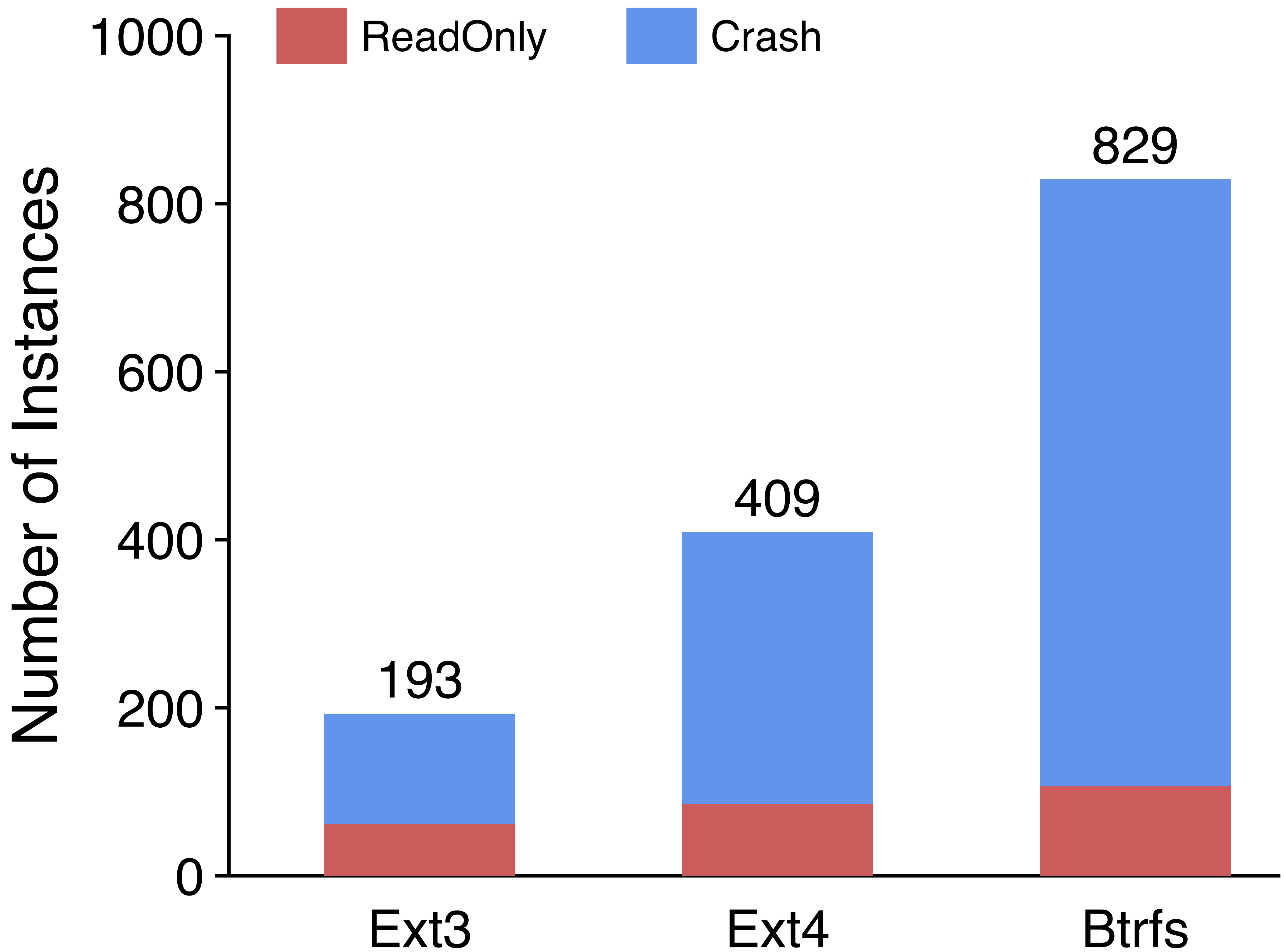
```
btrfs/disk-io.c, 3.8

open_ctree(...) {

    root->node = read_tree_block(...);
    BUG_ON(!root->node);
```

Crash Example

```
btrfs/disk-io.c, 3.8

open_ctree(...) {

    root->node = read_tree_block(...);
    BUG_ON(!root->node);
```

Crash Example

```
btrfs/disk-io.c, 3.8

open_ctree(...) {

    root->node = read_tree_block(...);
    BUG_ON(!root->node);
```


ReadOnly
Crash
Number of Instances
1000
800
600
400
200
0
193
409
829
Ext3
Ext4
Btrfs

Read-only and crash are common in modern file systems

Over 67% of global failures will crash the system

Q2:

What are the root causes of global failures ?

Global Failure Causes

Global Failure Causes

Metadata corruption

- metadata inconsistency is detected
- e.g., a block/inode bitmap corruption

Metadata Corruption Example

```
ext3/dir.c, 2.6.32

ext3_check_dir_entry(...){

 1    rlen = ext3_rec_len_from_disk();
 2    if (rlen < EXT3_DIR_REC_LEN(1)){
         error = “rec_len is too small”;
 3       ext3_error(sb, error);
      }
```

Metadata Corruption Example

```
ext3/dir.c, 2.6.32

ext3_check_dir_entry(...){

 1    rlen = ext3_rec_len_from_disk();
 2    if (rlen < EXT3_DIR_REC_LEN(1)){
         error = "rec_len is too small";
 3       ext3_error(sb, error);
      }
```

Metadata Corruption Example

```
ext3/dir.c, 2.6.32

ext3_check_dir_entry(...){

 1    rlen = ext3_rec_len_from_disk();
 2    if (rlen < EXT3_DIR_REC_LEN(1)){
          error = "rec_len is too small";
 3        ext3_error(sb, error);
      }
```

Metadata Corruption Example

```
ext3/dir.c, 2.6.32

ext3_check_dir_entry(...){

1     rlen = ext3_rec_len_from_disk();
2     if (rlen < EXT3_DIR_REC_LEN(1)){
          error = "rec_len is too small";
3         ext3_error(sb, error);
      }
```

Global Failure Causes

Metadata corruption

- metadata inconsistency is detected
- e.g., a block/inode bitmap corruption

I/O failure

- metadata I/O failure and journaling failure
- e.g., fail to read an inode block

I/O Failure Example

```
ext4/namei.c, 2.6.32

empty_dir(...){

1    bh = ext4_bread(NULL, inode, &err);
     if (bh && err)
2       EXT4_ERROR_INODE(inode,
         “fail to read directory block”);
```

I/O Failure Example

```
ext4/namei.c, 2.6.32

empty_dir(...){

1     bh = ext4_bread(NULL, inode, &err);
      if (bh && err)
2         EXT4_ERROR_INODE(inode,
           "fail to read directory block");
```

I/O Failure Example

```
ext4/namei.c, 2.6.32

empty_dir(...){

 1    bh = ext4_bread(NULL, inode, &err);
      if (bh && err)
 2        EXT4_ERROR_INODE(inode,
            "fail to read directory block");
```

I/O Failure Example

```
ext4/namei.c, 2.6.32

empty_dir(...){

1    bh = ext4_bread(NULL, inode, &err);
     if (bh && err)
2        EXT4_ERROR_INODE(inode,
          “fail to read directory block”);
```

Global Failure Causes

Metadata corruption

- metadata inconsistency is detected
- e.g., a block/inode bitmap corruption

I/O failure

- metadata I/O failure and journaling failure
- e.g., fail to read an inode block

Software bugs

- unexpected states detected
- e.g., allocated block is not in a valid range

Software Bug Example

```
ext3/balloc.c, 2.6.32

ext3_rsv_window_add(...){

 1    if (start < this->rsv_start)
          p = &(*p)->rb->left;
 2    else if (start > this->rsv_end)
          p = &(*p)->rb->right;
 3    else {
         rsv_window_dump(root, 1);
 4        BUG();
      }
```

Software Bug Example

```
ext3/balloc.c, 2.6.32

ext3_rsv_window_add(...){

1    if (start < this->rsv_start)
         p = &(*p)->rb->left;
2    else if (start > this->rsv_end)
         p = &(*p)->rb->right;
3    else {
        rsv_window_dump(root, 1);
4        BUG();
     }
```

Software Bug Example

```
ext3/balloc.c, 2.6.32

ext3_rsv_window_add(...){

 1    if (start < this->rsv_start)
          p = &(*p)->rb->left;
 2    else if (start > this->rsv_end)
          p = &(*p)->rb->right;
 3    else {
          rsv_window_dump(root, 1);
 4        BUG();
      }
```

Software Bug Example

```
ext3/balloc.c, 2.6.32

ext3_rsv_window_add(...){

 1    if (start < this->rsv_start)
          p = &(*p)->rb->left;
 2    else if (start > this->rsv_end)
          p = &(*p)->rb->right;
 3    else {
          rsv_window_dump(root, 1);
 4        BUG();
      }
```

Software Bug Example

```
ext3/balloc.c, 2.6.32

ext3_rsv_window_add(...){

 1    if (start < this->rsv_start)
          p = &(*p)->rb->left;
 2    else if (start > this->rsv_end)
          p = &(*p)->rb->right;
 3    else {
          rsv_window_dump(root, 1);
 4        BUG();
      }
```

Ext3

Data Structure	MC	IOF	SB	Shared
b-bitmap	2	2		Yes
i-bitmap	1	1		Yes
inode	1	2	2	Yes
super	1			Yes
dir-entry	4	4	3	Yes
gdt	3		2	Yes
indir-blk	1	1		**No**
xattr	5	2	1	**No**
block			5	Yes/No
journal		3	27	Yes
journal_head			31	Yes
buf_head			16	Yes
handle		22	9	Yes
transaction			28	Yes
revoke			2	Yes
other	1		11	Yes/No
Total	19	37	137	= 193

Ext3

Data Structure	**MC**	**IOF**	**SB**	**Shared**
b-bitmap	2	2		Yes
i-bitmap	1	1		Yes
inode	1	2	2	Yes
super	1			Yes
dir-entry	4	4	3	Yes
gdt	3		2	Yes
indir-blk	1	1		**No**
xattr	5	2	1	**No**
block			5	Yes/No
journal		3	27	Yes
journal_head			31	Yes
buf_head			16	Yes
handle		22	9	Yes
transaction			28	Yes
revoke			2	Yes
other	1		11	Yes/No
Total	19	37	137	= 193

Ext3

Data Structure	**MC**	**IOF**	**SB**	**Shared**
b-bitmap	2	2		Yes
i-bitmap	1	1		Yes
inode	1	2	2	Yes
super	1			Yes
dir-entry	4	4	3	Yes
gdt	3		2	Yes
indir-blk	1	1		**No**
xattr	5	2	1	**No**
block			5	Yes/No
journal		3	27	Yes
journal_head			31	Yes
buf_head			16	Yes
handle		22	9	Yes
transaction			28	Yes
revoke			2	Yes
other	1		11	Yes/No
Total	19	37	137	= 193

Ext3

Data Structure	MC	IOF	SB	Shared
b-bitmap	2	2		Yes
i-bitmap	1	1		Yes
inode	1	2	2	Yes
super	1			Yes
dir-entry	4	4	3	Yes
gdt	3		2	Yes
indir-blk	1	1		**No**
xattr	5	2	1	**No**
block			5	Yes/No
journal		3	27	Yes
journal_head			31	Yes
buf_head			16	Yes
handle		22	9	Yes
transaction			28	Yes
revoke			2	Yes
other	1		11	Yes/No
Total	19	37	137	= 193

Ext3

Data Structure	**MC**	**IOF**	**SB**	**Shared**
b-bitmap	2	2		Yes
i-bitmap	1	1		Yes
inode	1	2	2	Yes
super	1			Yes
dir-entry	4	4	3	Yes
gdt	3		2	Yes
indir-blk	1	1		**No**
xattr	5	2	1	**No**
block			5	Yes/No
journal		3	27	Yes
journal_head			31	Yes
buf_head			16	Yes
handle		22	9	Yes
transaction			28	Yes
revoke			2	Yes
other	1		11	Yes/No
Total	19	37	137	= 193

Ext3

Data Structure	MC	IOF	SB	Shared
b-bitmap	2	2		Yes
i-bitmap	1	1		Yes
inode	1	2	2	Yes
super	1			Yes
dir-entry	4	4	3	Yes
gdt	3		2	Yes
indir-blk	1	1		**No**
xattr	5	2	1	**No**
block			5	Yes/No
journal		3	27	Yes
journal_head			31	Yes
buf_head			16	Yes
handle		22	9	Yes
transaction			28	Yes
revoke			2	Yes
other	1		11	Yes/No
Total	19	37	137	= 193

Ext3

Data Structure	MC	IOF	SB	Shared
b-bitmap	2	2		Yes
i-bitmap	1	1		Yes
inode	1	2	2	Yes
super	1			Yes
dir-entry	4	4	3	Yes
gdt	3		2	Yes
indir-blk	1	1		**No**
xattr	5	2	1	**No**
block			5	Yes/No
journal		3	27	Yes
journal_head			31	Yes
buf_head			16	Yes
handle		22	9	Yes
transaction			28	Yes
revoke			2	Yes
other	1		11	Yes/No
Total	19	37	137	= 193

All global failures are caused by **metadata** and **system states**

All global failures are caused by **metadata** and **system states**

Both **local** and **shared** metadata can cause global failures

All global failures are caused by **metadata** and **system states**

Both **local** and **shared** metadata can cause global failures

Not Only Local File Systems

Not Only Local File Systems

Shared-disk file systems OCFS2

- inspired by Ext3 design
- used in virtualization environment
  - host virtual machine images
  - allow multiple Linux guests to share a file system

Not Only Local File Systems

Shared-disk file systems OCFS2

- inspired by Ext3 design
- used in virtualization environment
  - host virtual machine images
  - allow multiple Linux guests to share a file system

Global failures are also prevalent

- a single piece of corrupted metadata can fail the whole file system on multiple nodes !

Current Abstractions

Current Abstractions

File and directory

- metadata is shared for different files or directories

Current Abstractions

File and directory

- metadata is shared for different files or directories

Namespace

- virtual machines, Chroot, BSD jail, Solaris Zones
- multiple namespaces still share a file system

Current Abstractions

File and directory

- metadata is shared for different files or directories

Namespace

- virtual machines, Chroot, BSD jail, Solaris Zones
- multiple namespaces still share a file system

Partitions

- multiple file systems on separated partitions
- a single panic on a partition can crash the whole operating system
- static partitions, dynamic partitions
- management of many partitions

All files on a file system implicitly share

a single fault domain

All files on a file system implicitly share

a single fault domain

All files on a file system implicitly share
a single fault domain

Current file-system abstractions do not
provide fine-grained fault isolation

Introduction

Study of Failure Policies

Isolation File Systems

- New Abstraction
- Fault Isolation
- Quick Recovery
- Preliminary Implementation on Ext3

Challenges

Isolation File Systems

Isolation File Systems

Fine-grained partitioned

- files are isolated into separated domains

Isolation File Systems

Fine-grained partitioned

- files are isolated into separated domains

Independent

- faulty units will not affect healthy units

Isolation File Systems

Fine-grained partitioned

- files are isolated into separated domains

Independent

- faulty units will not affect healthy units

Fine-grained recovery

- repair a faulty unit quickly
- instead of checking the whole file system

Isolation File Systems

Fine-grained partitioned

- files are isolated into separated domains

Independent

- faulty units will not affect healthy units

Fine-grained recovery

- repair a faulty unit quickly
- instead of checking the whole file system

Elastic

- dynamically grow and shrink its size

New Abstraction

New Abstraction

File Pod

- an abstract partition
- contains a group of files and *related metadata*
- an independent fault domain

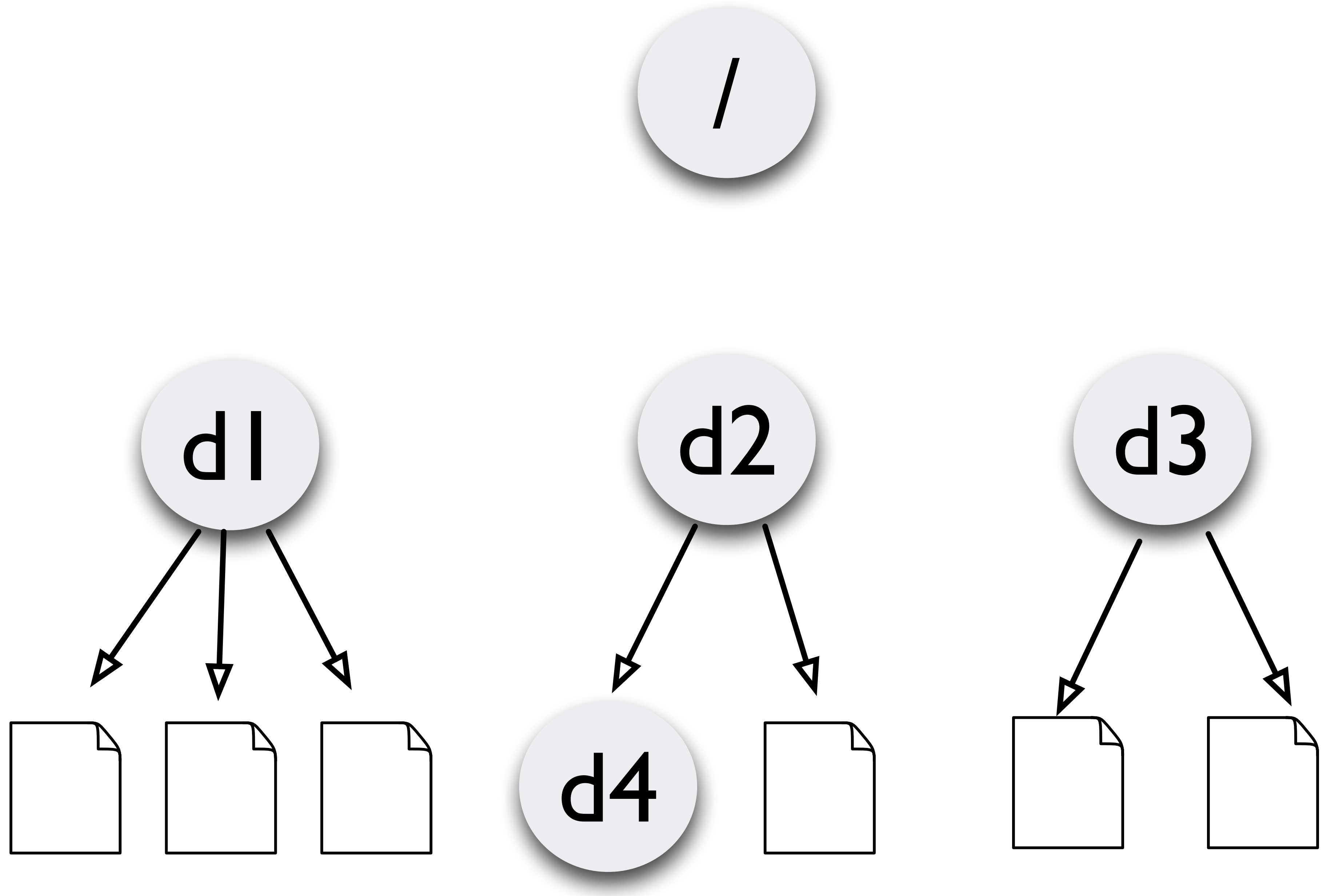
New Abstraction

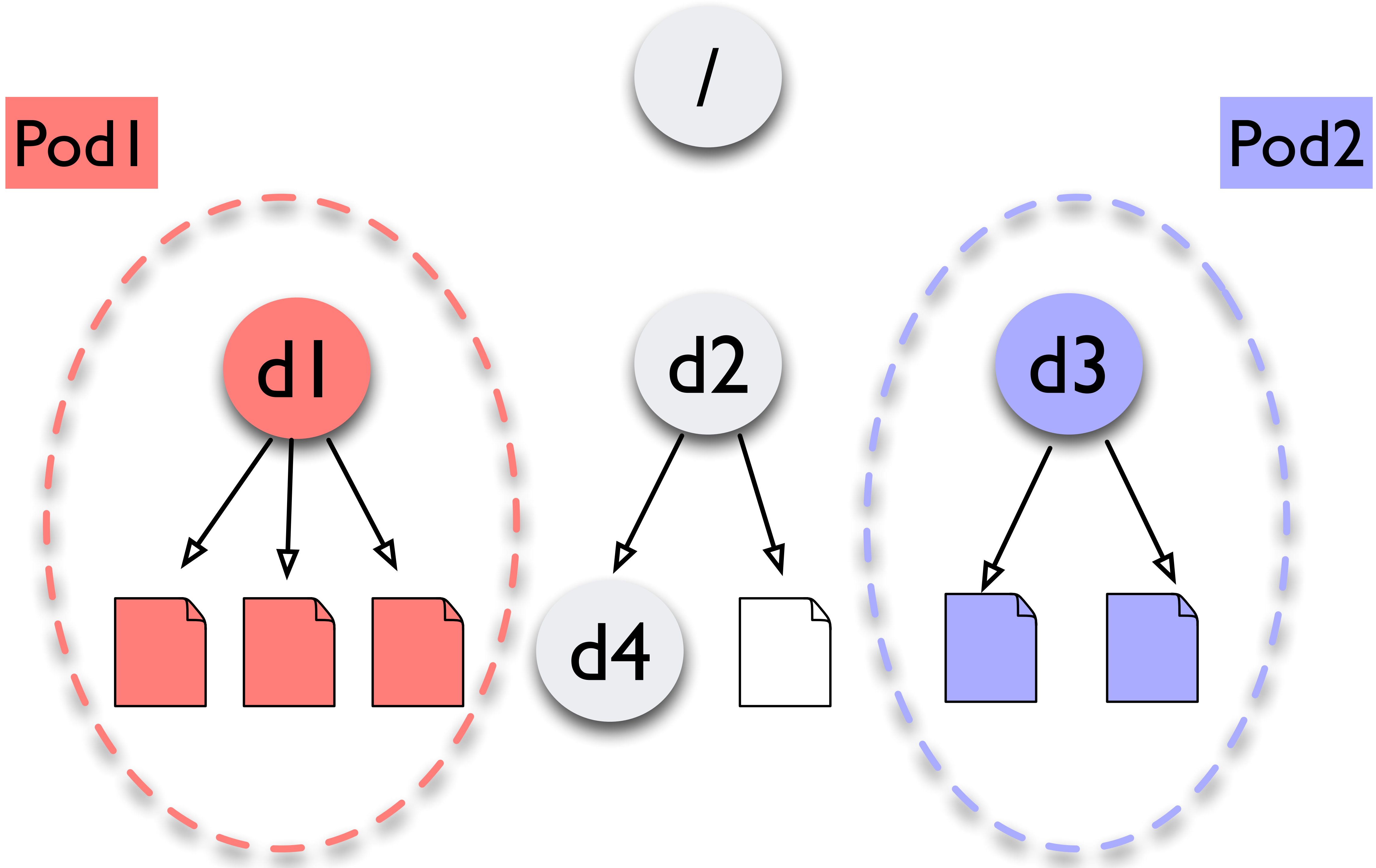
File Pod

- an abstract partition
- contains a group of files and *related metadata*
- an independent fault domain

Operations

- create a file pod
- set / get file pod's attributes
  - failure policy
  - recovery policy
- bind / unbind a file to pod
- share a file between pods


/
d1
d2
d3
d4


/
Pod1
Pod2
d1
d2
d3
d4

Introduction

Study of Failure Policies

Isolation File Systems

- New Abstraction
- Fault Isolation
- Quick Recovery
- Preliminary Implementation on Ext3

Challenges

Metadata Isolation

Metadata Isolation

Observation

- metadata is organized in a shared manner
- hard to isolate a failure for metadata

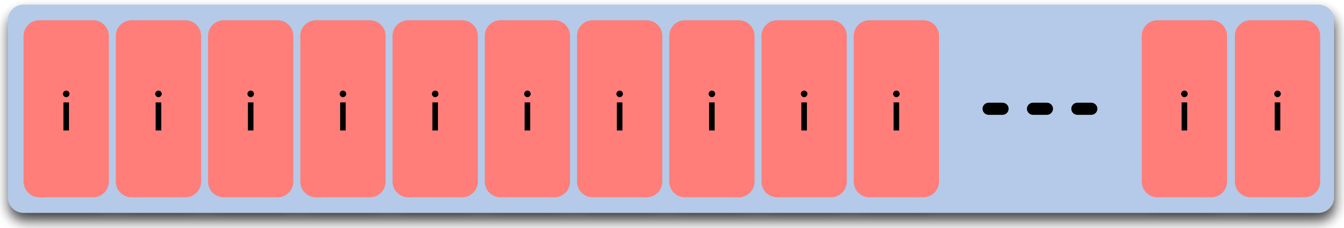
Metadata Isolation

Observation

- ➡ metadata is organized in a shared manner
- ➡ hard to isolate a failure for metadata

For example

- ➡ multiple inodes are stored in a single inode block



an inode block

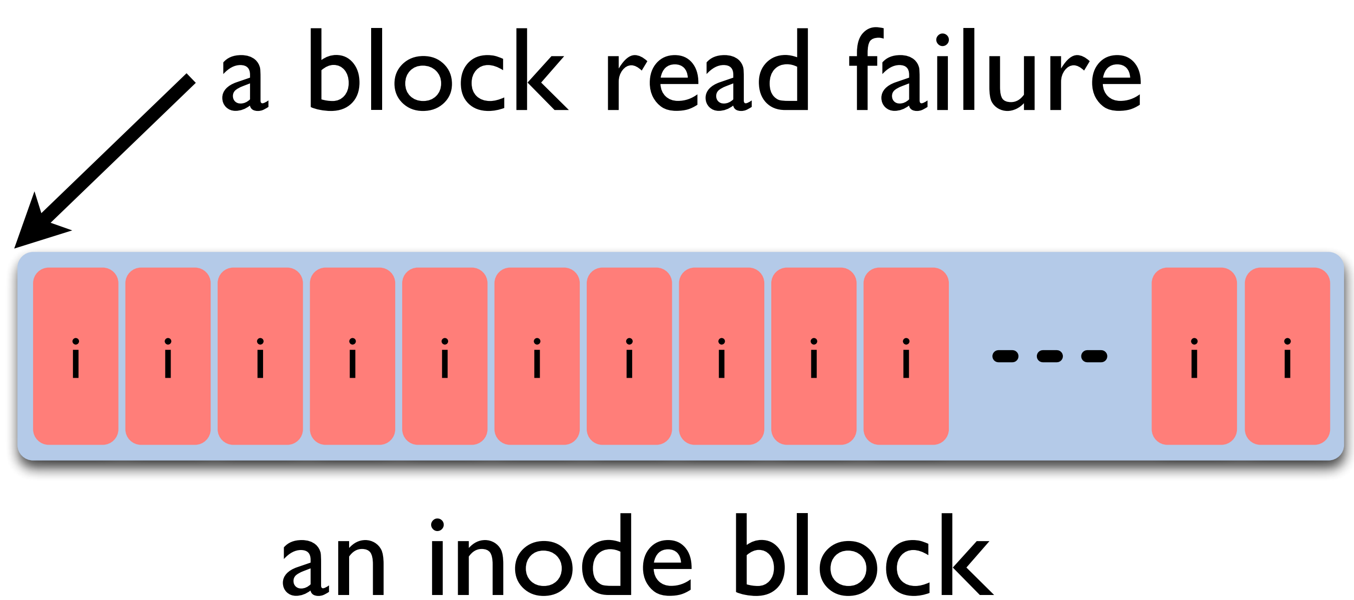
Metadata Isolation

Observation

- metadata is organized in a shared manner
- hard to isolate a failure for metadata

For example

- multiple inodes are stored in a single inode block
- an I/O failure can affect multiple files



Key Idea 1:

Key Idea 1:

Isolate metadata for file pods

Localize Failures

Localize Failures

Local Failures

- convert global failures to local failures
- same failure semantics
- only fail the faulty pod

Localize Failures

Local Failures

- convert global failures to local failures
- same failure semantics
- only fail the faulty pod

Read-Only

- mark a file pod as Read-Only

Localize Failures

Local Failures

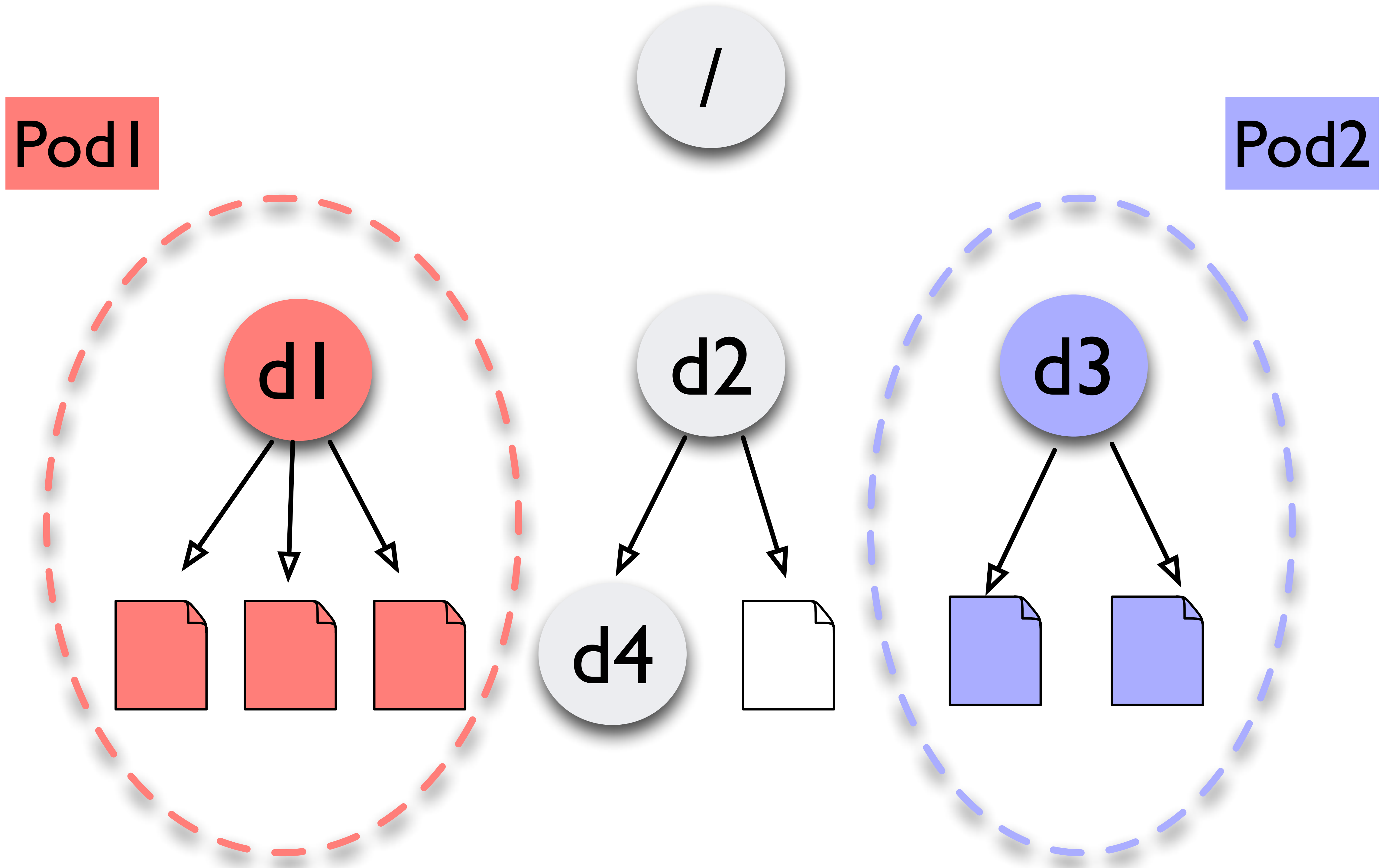
- convert global failures to local failures
- same failure semantics
- only fail the faulty pod

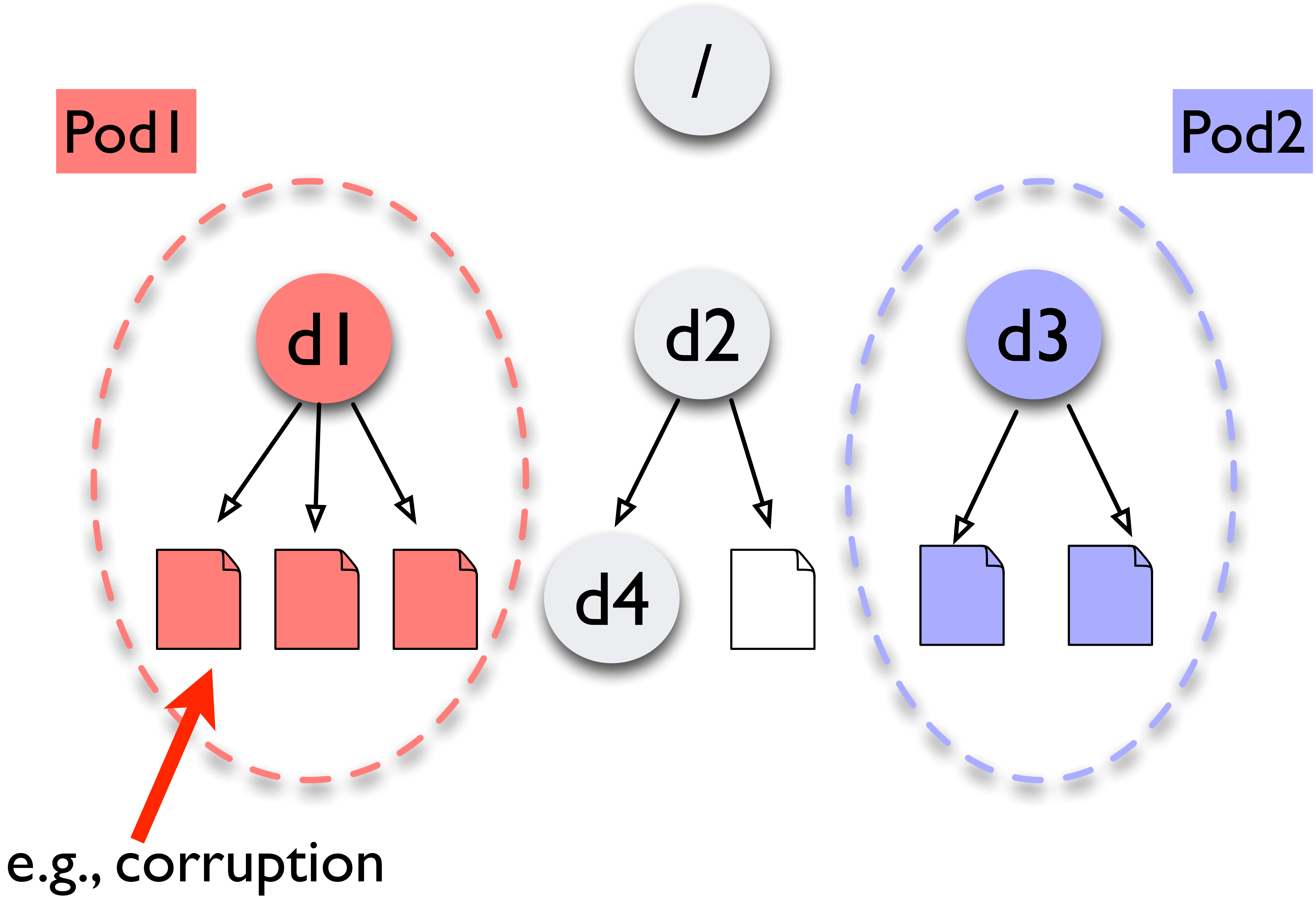
Read-Only

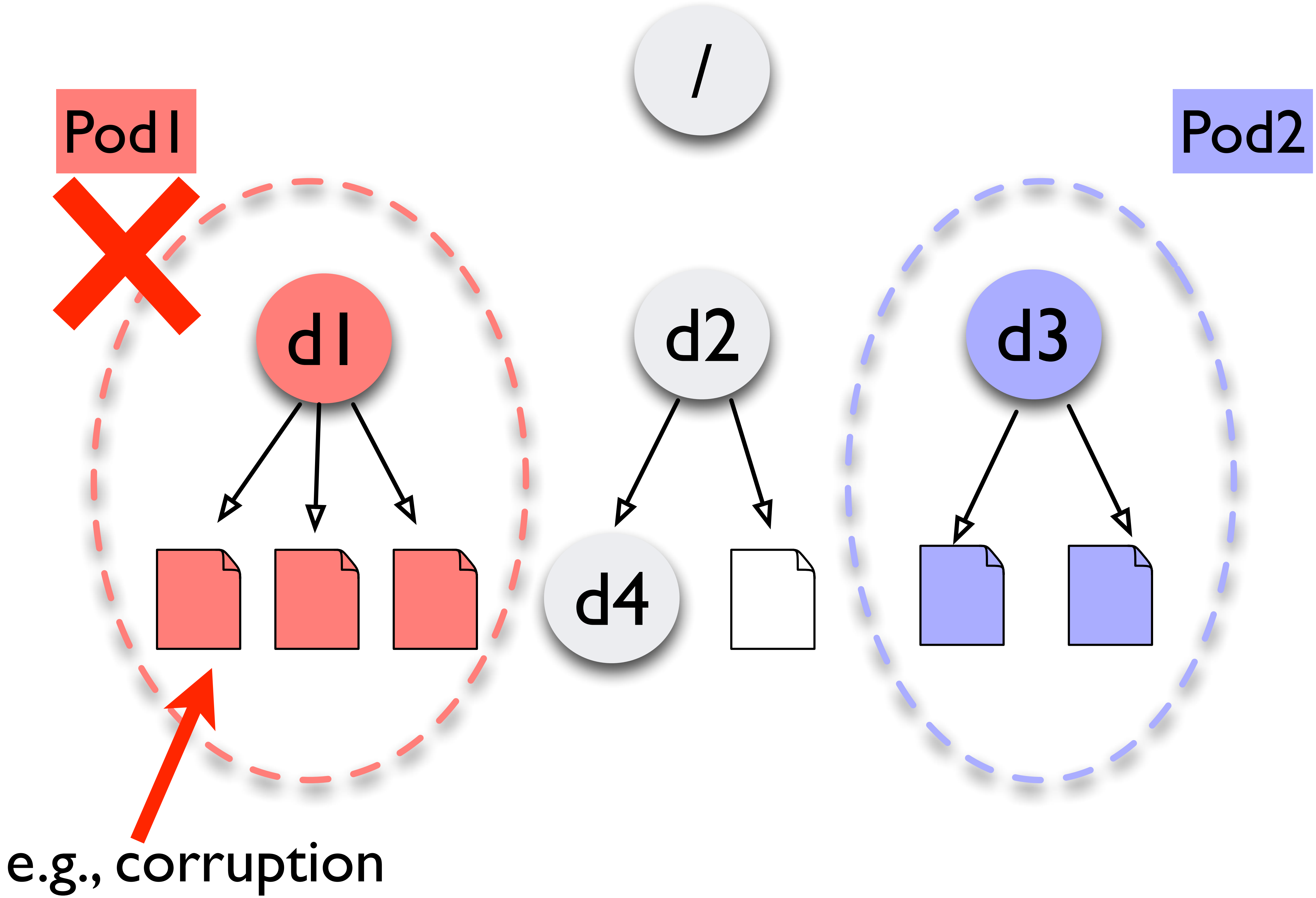
- mark a file pod as Read-Only

Crash

- crash a file pod instead of the whole system
- provide the same initial states after crash


/
Pod1
Pod2
d1
d2
d3
d4


/
Pod1
Pod2
d1
d2
d3
d4
e.g., corruption


/
Pod1
Pod2
d1
d2
d3
d4
e.g., corruption

Introduction

Study of Failure Policies

Isolation File Systems

- New Abstraction
- Fault Isolation
- Quick Recovery
- Preliminary Implementation on Ext3

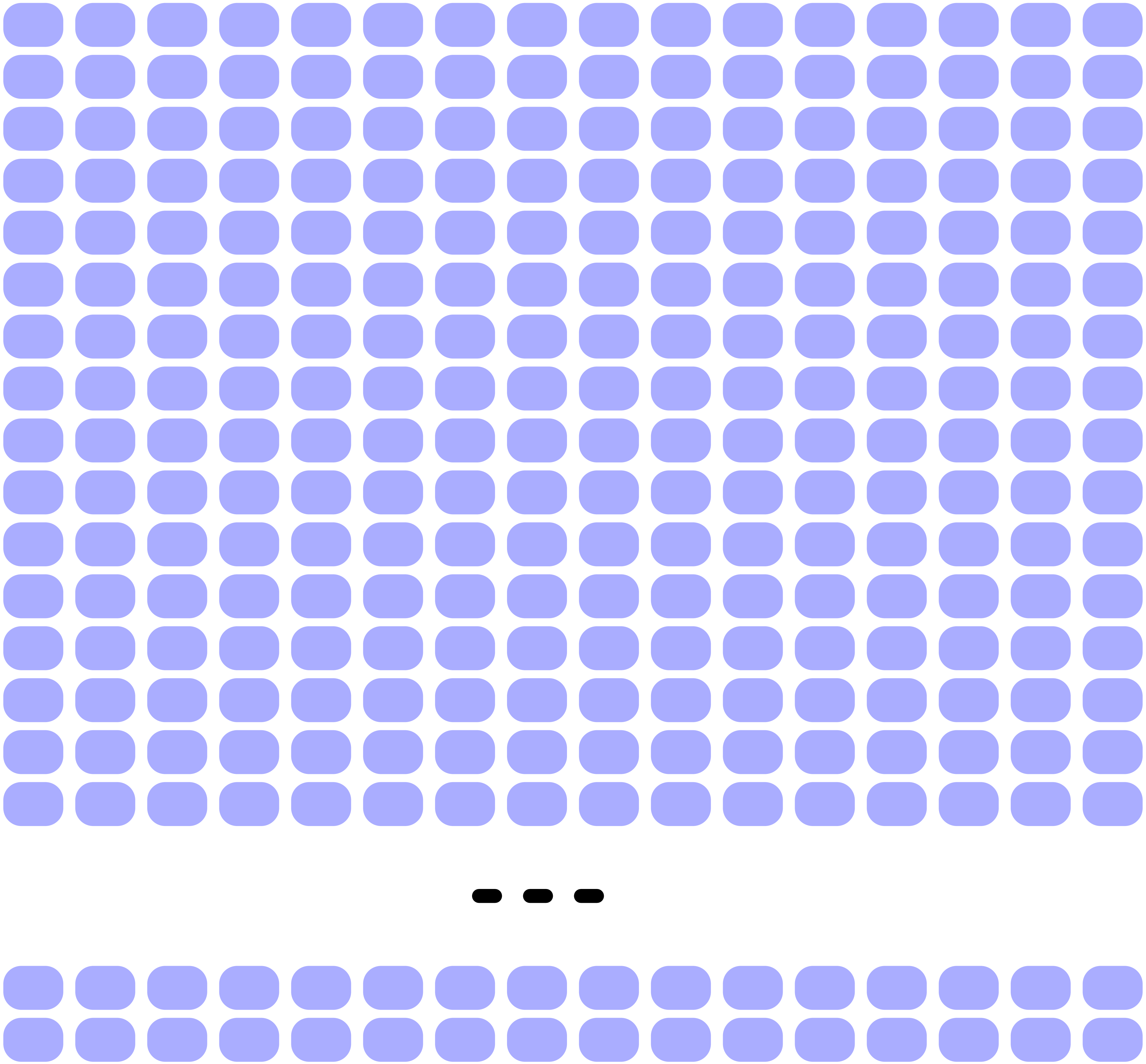
Challenges

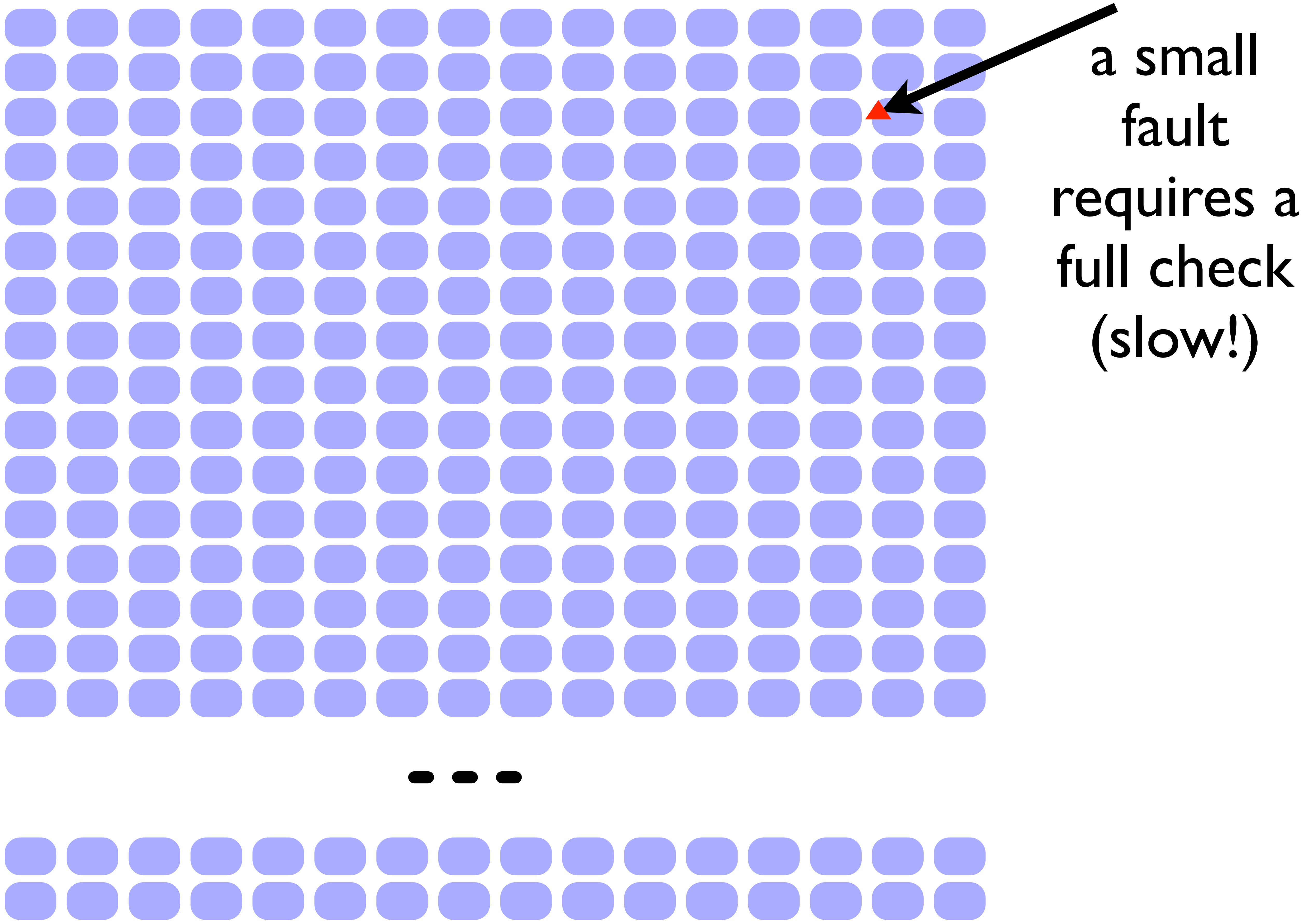
Quick Recovery

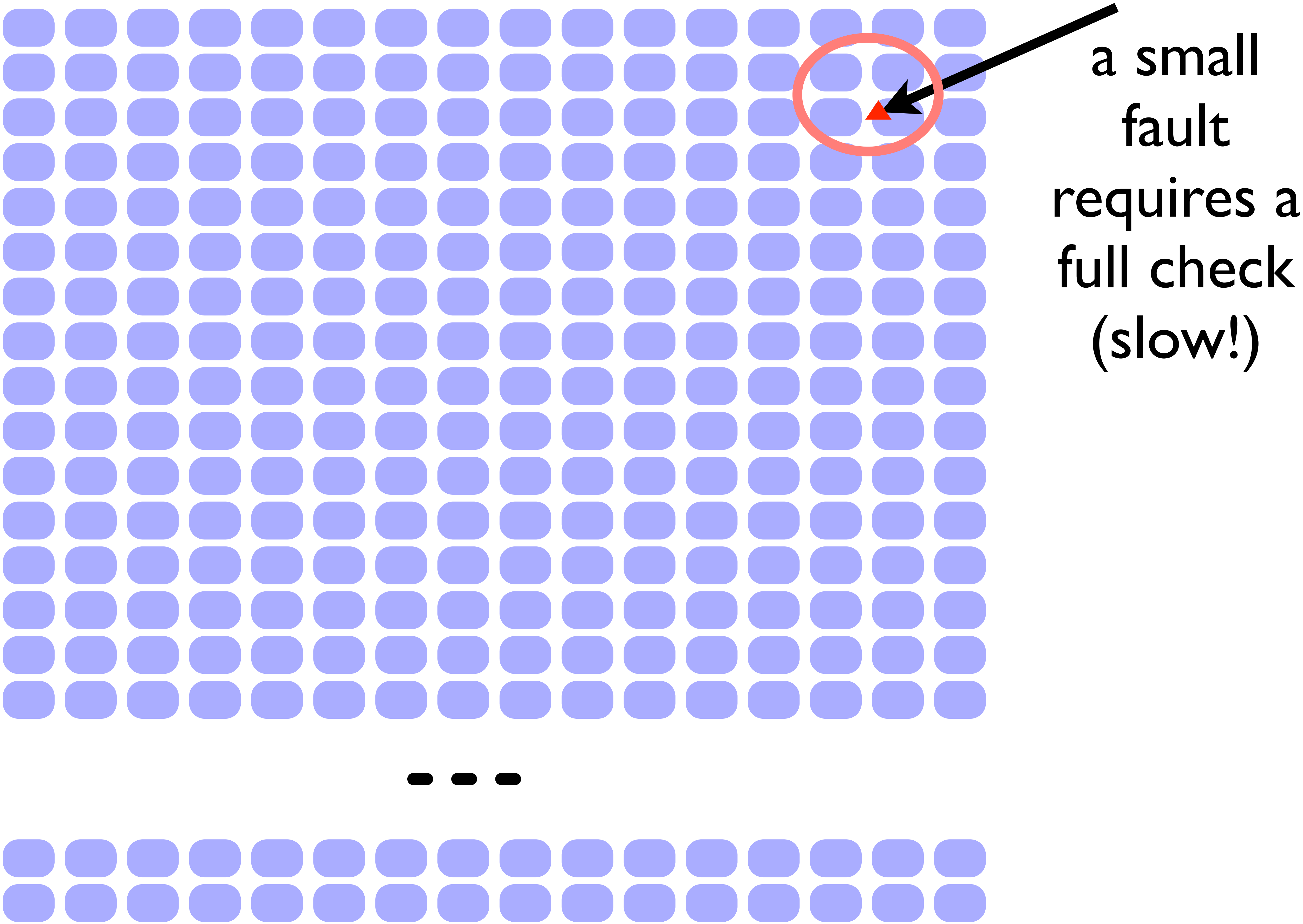
Quick Recovery

File system recovery is slow

- a small error requires a full check
- many random read requests
- 7 hours to sequentially read a 2 TB disk




a small fault requires a full check (slow!)


a small fault requires a full check (slow!)

Key Idea 2:

Key Idea 2:

Minimize the file system checking range during recovery

Quick Recovery

Quick Recovery

Metadata Isolation

- file pod as the unit of recovery
- check and recover independently
- both online and offline

Quick Recovery

Metadata Isolation

- file pod as the unit of recovery
- check and recover independently
- both online and offline

When recover ?

- leverage internal detection mechanism

Quick Recovery

Metadata Isolation

- file pod as the unit of recovery
- check and recover independently
- both online and offline

When recover ?

- leverage internal detection mechanism

How to recover more efficiently ?

- only check the faulty pod
- narrow down to certain data structures

Introduction

Study of Failure Policies

Isolation File Systems

- New Abstraction
- Fault Isolation
- Quick Recovery
- Preliminary Implementation on Ext3

Challenges

Ext3 Layout

Ext3 Layout

A disk is divided into block groups

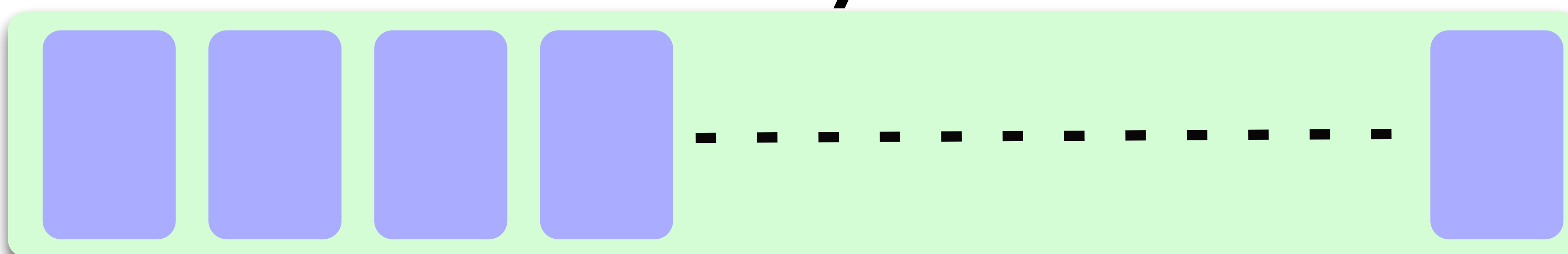
- ➡ physical partition for disk locality

Ext3 Layout

A disk is divided into block groups

- physical partition for disk locality

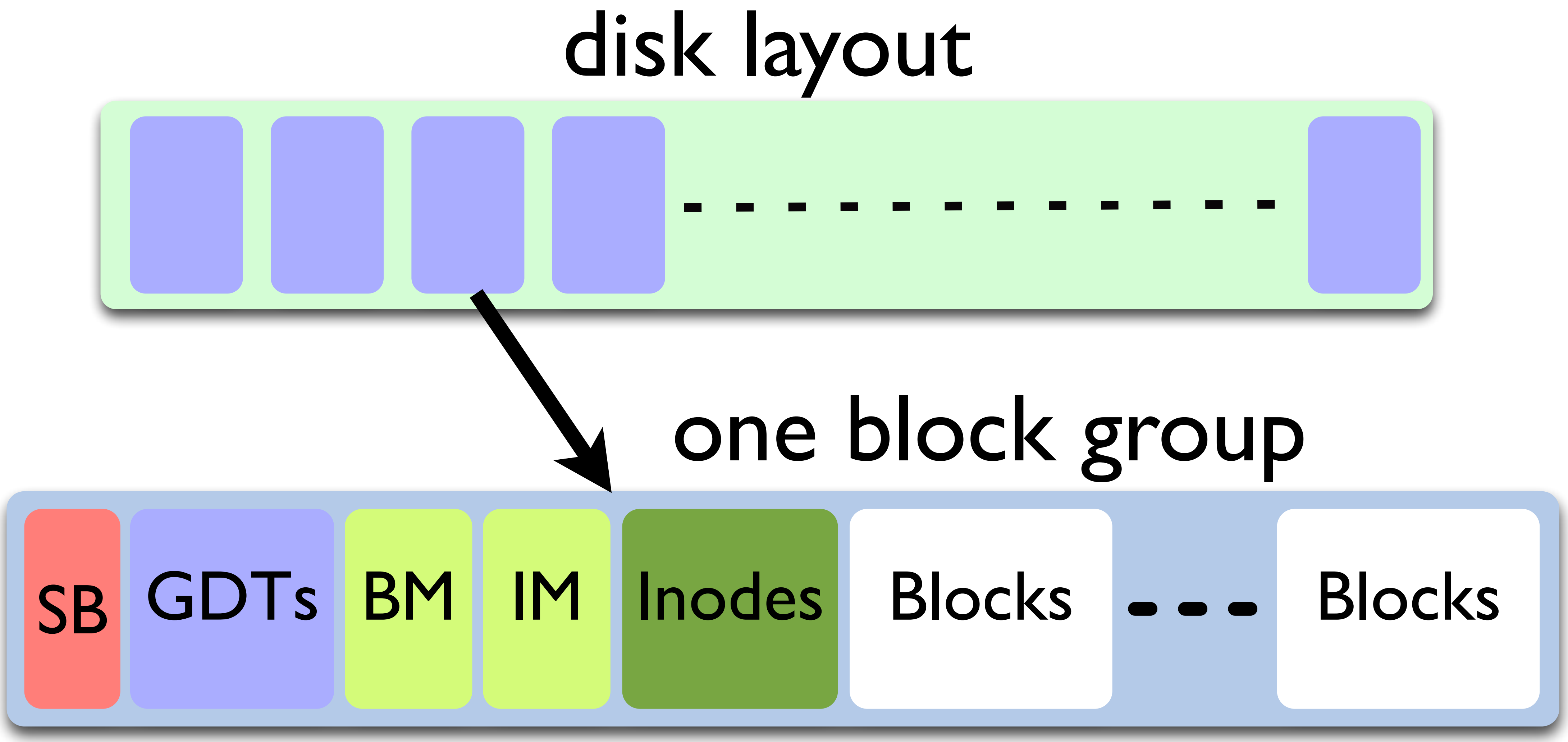
disk layout

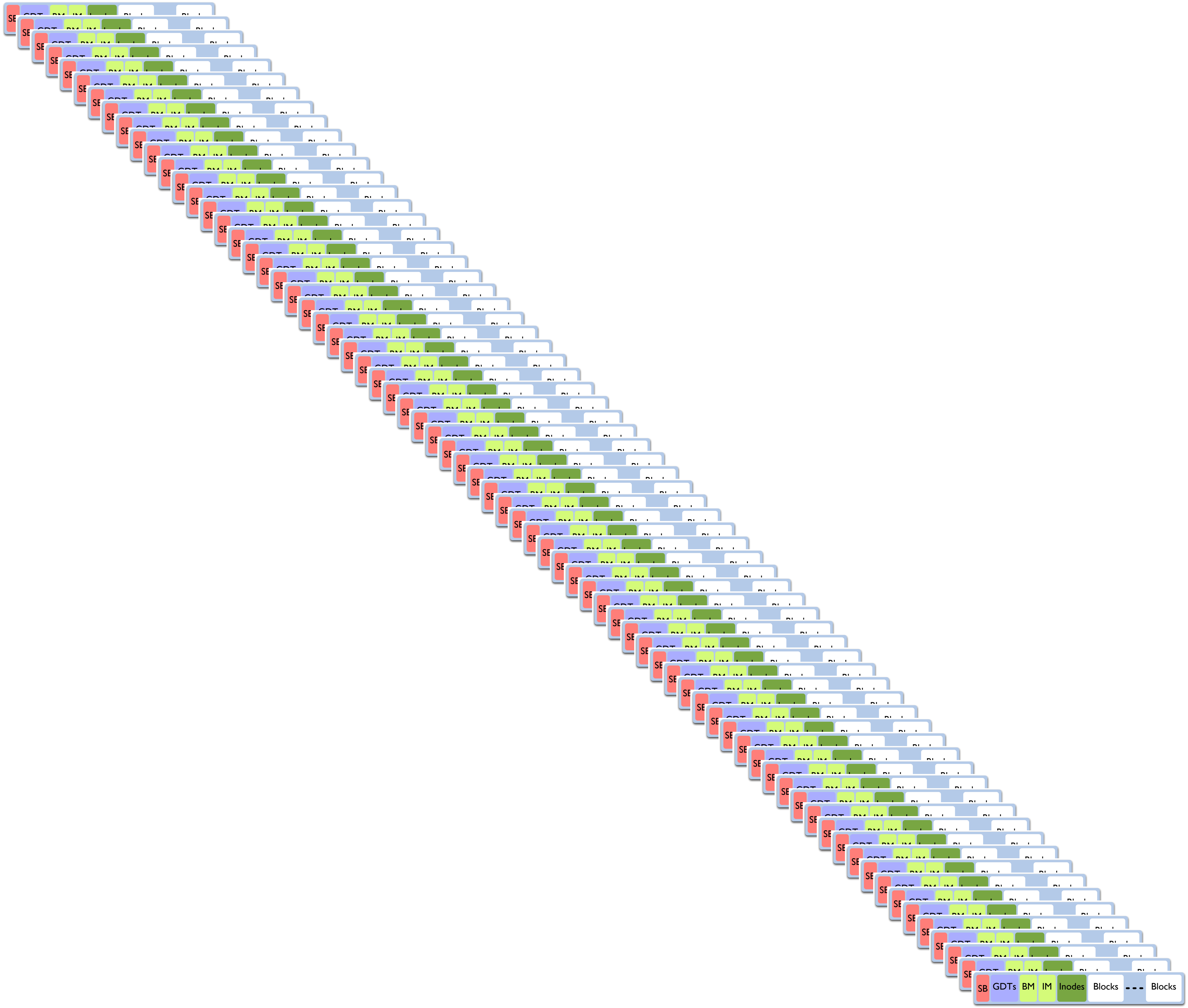


Ext3 Layout

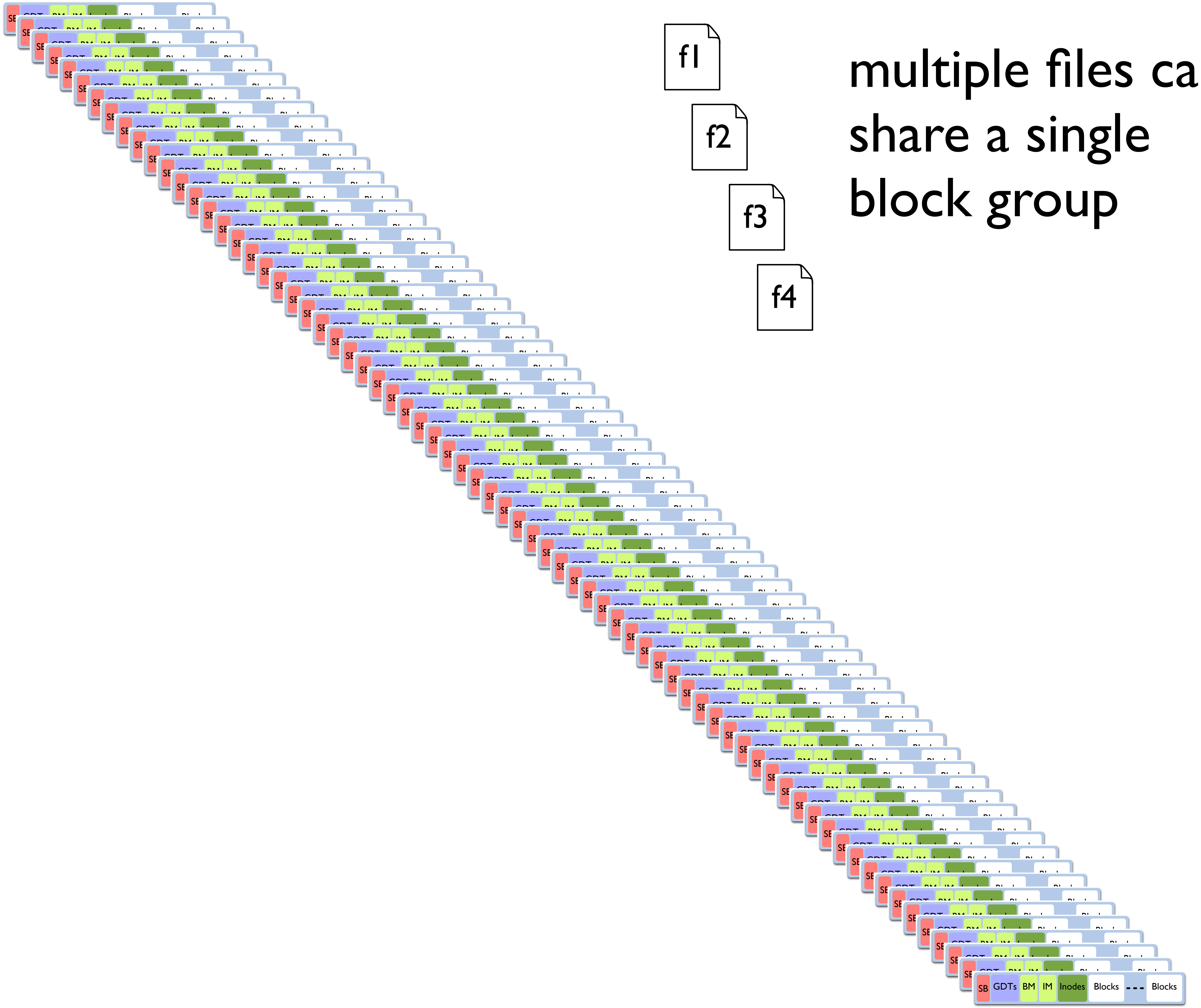
A disk is divided into block groups

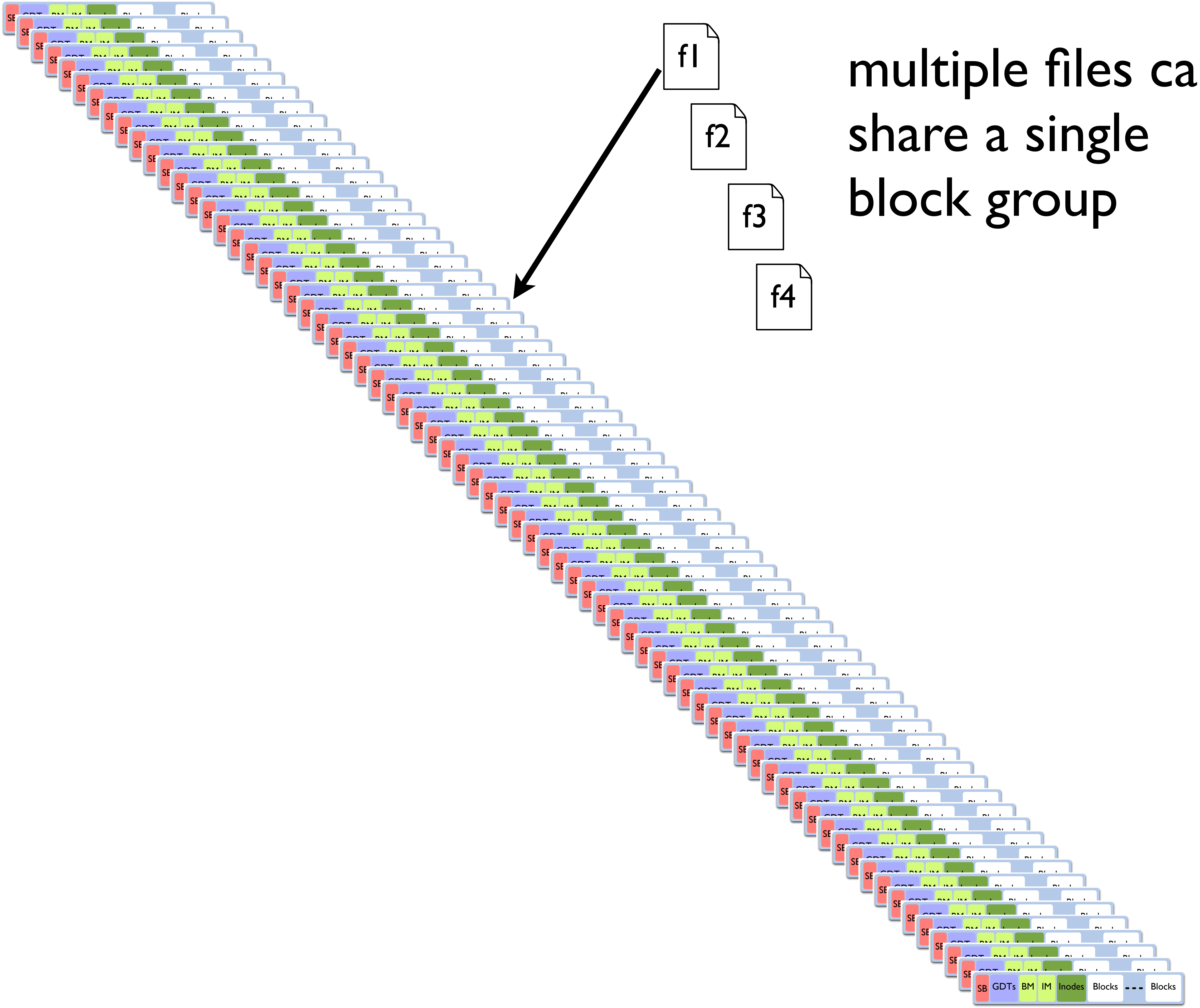
- physical partition for disk locality




SB
GDTs
BM
IM
Inodes
Blocks
Blocks

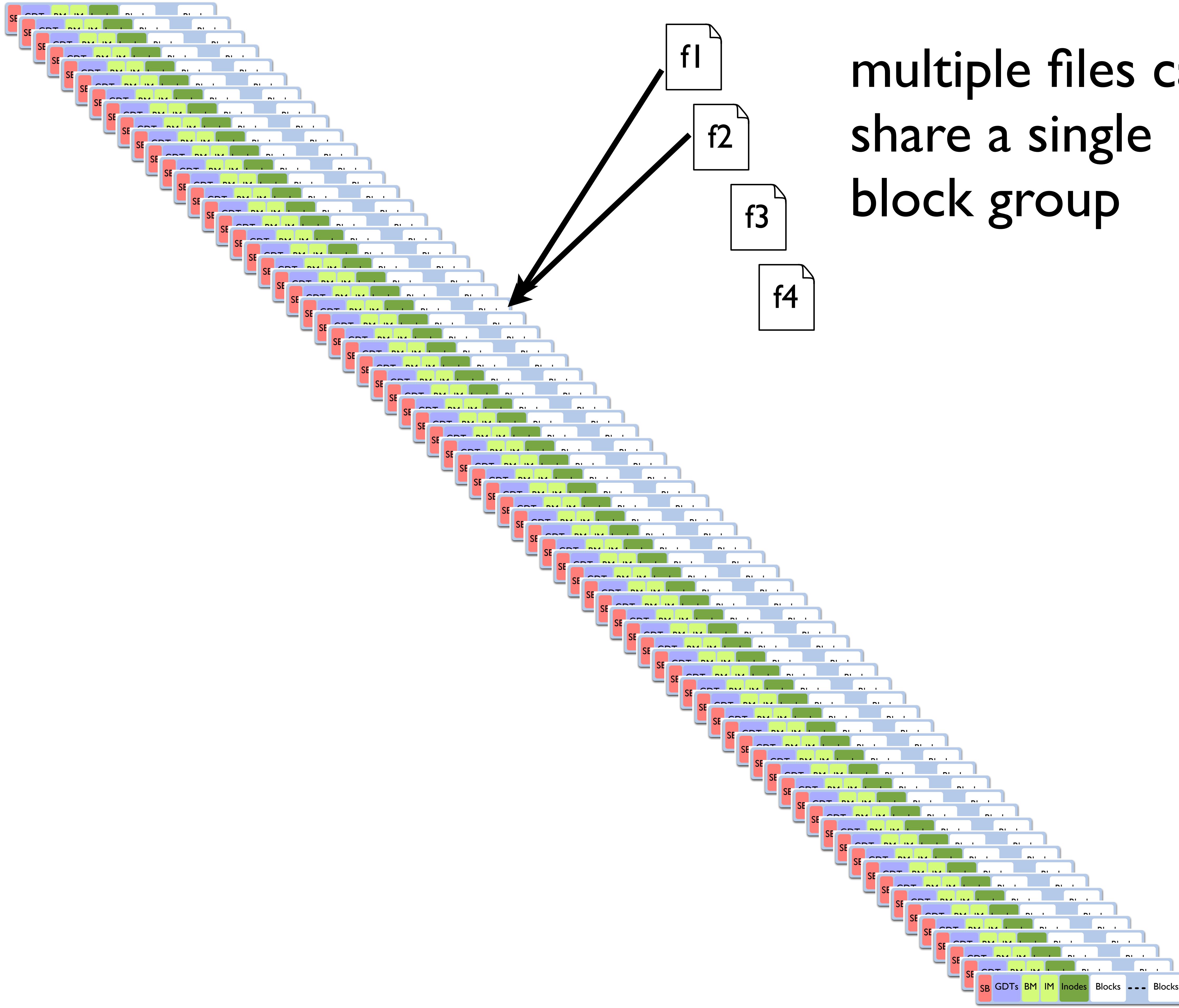
multiple files can share a single block group

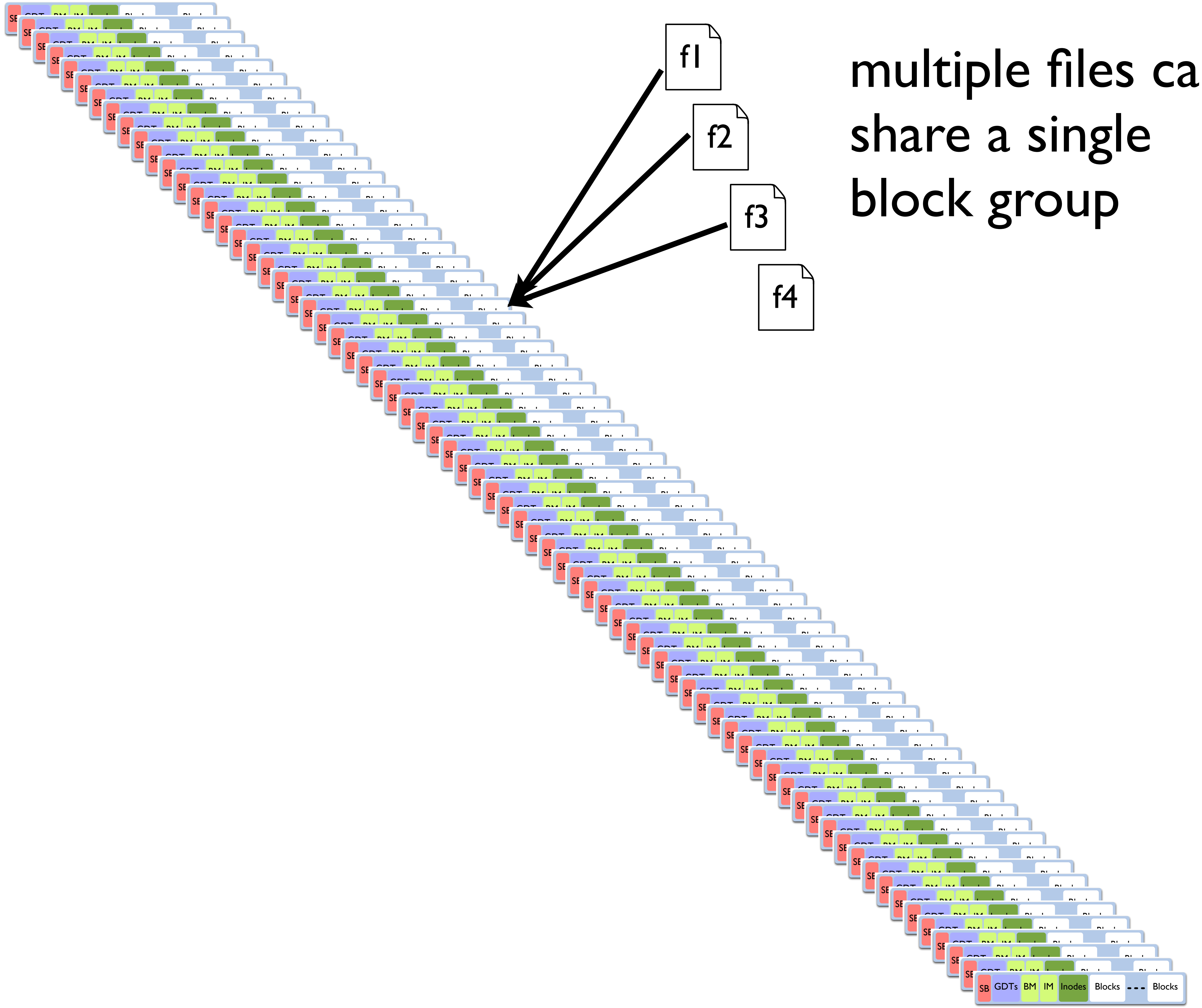




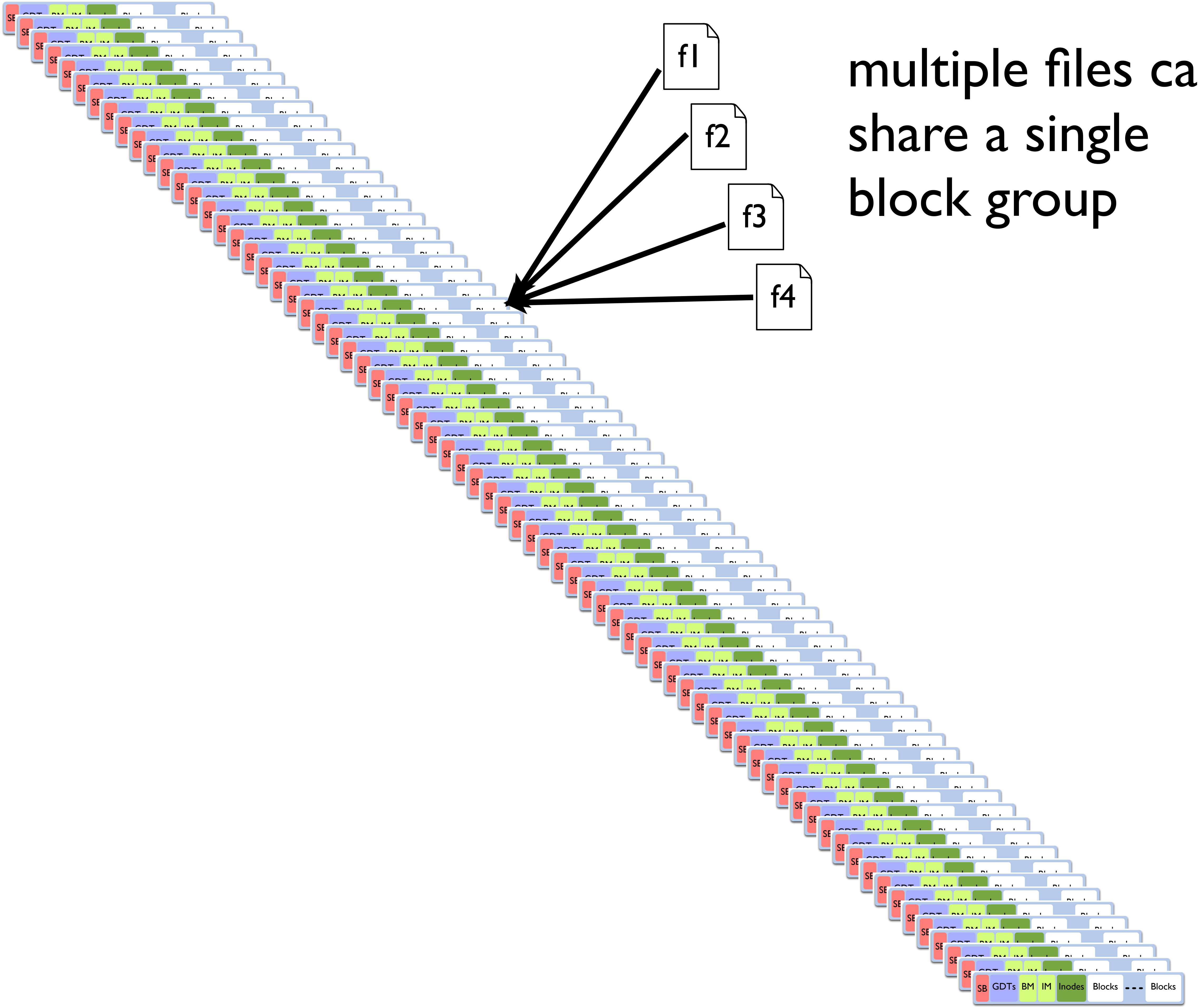
multiple files can share a single block group

multiple files can share a single block group

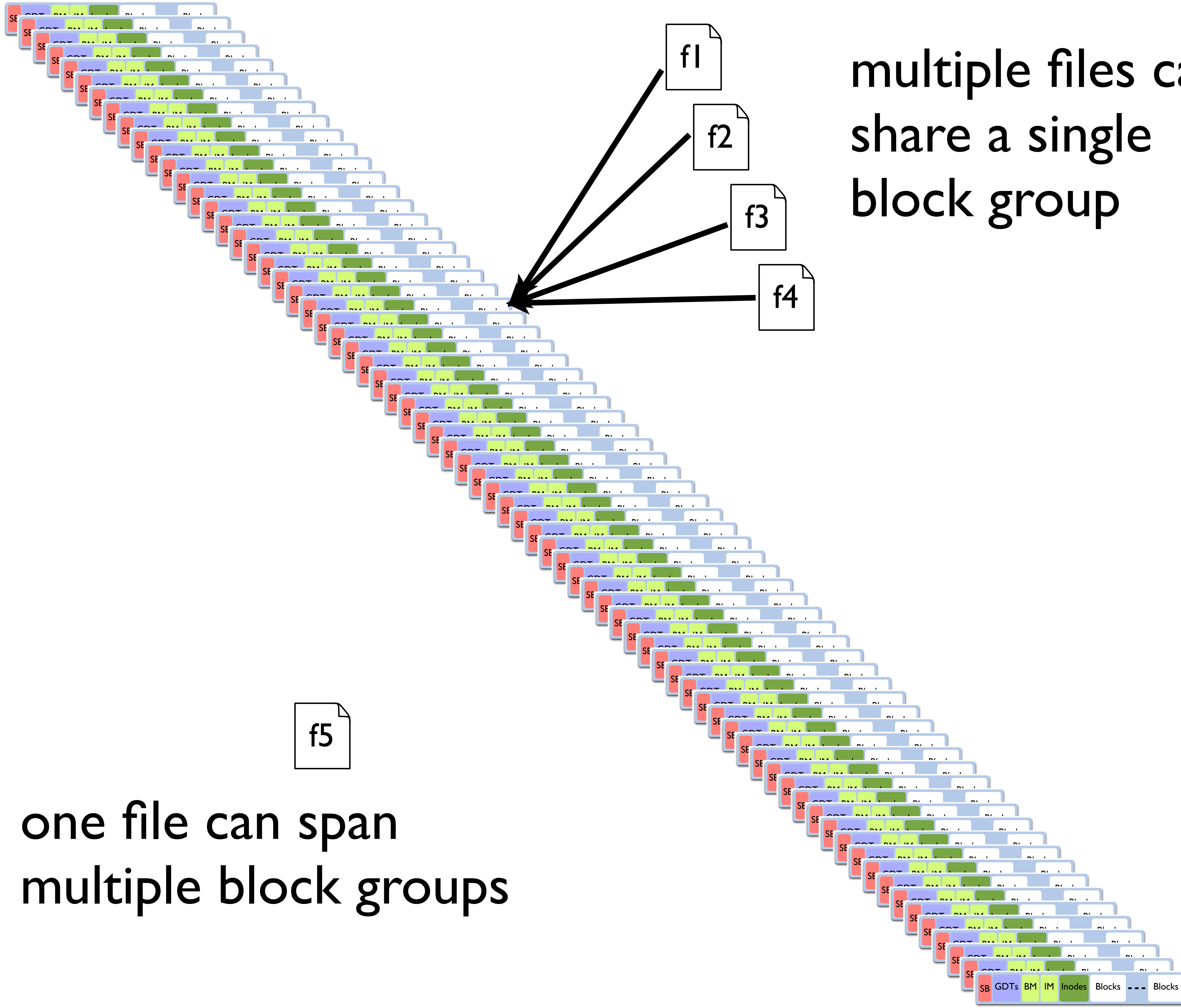




multiple files can share a single block group

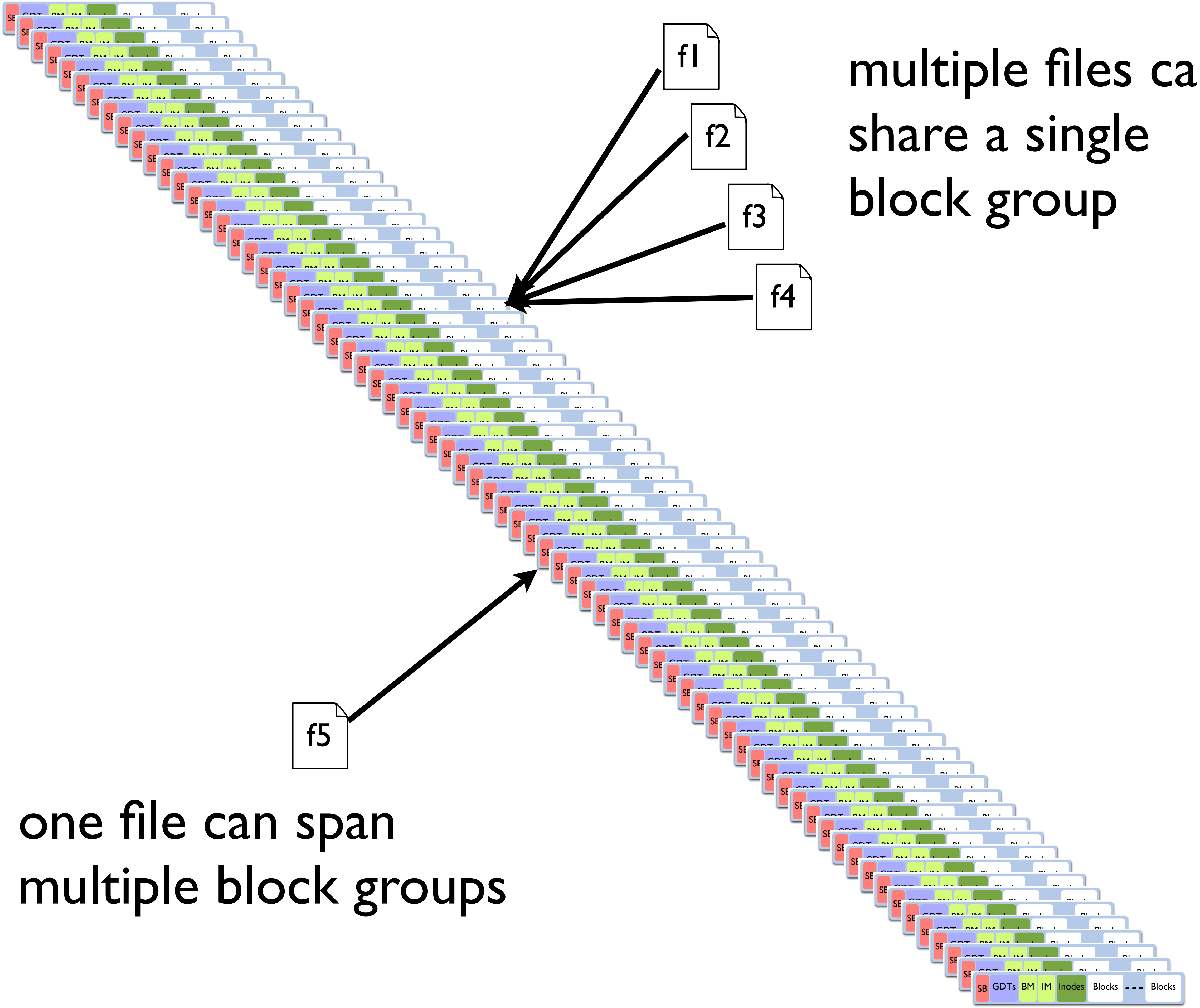


multiple files can share a single block group



multiple files can share a single block group

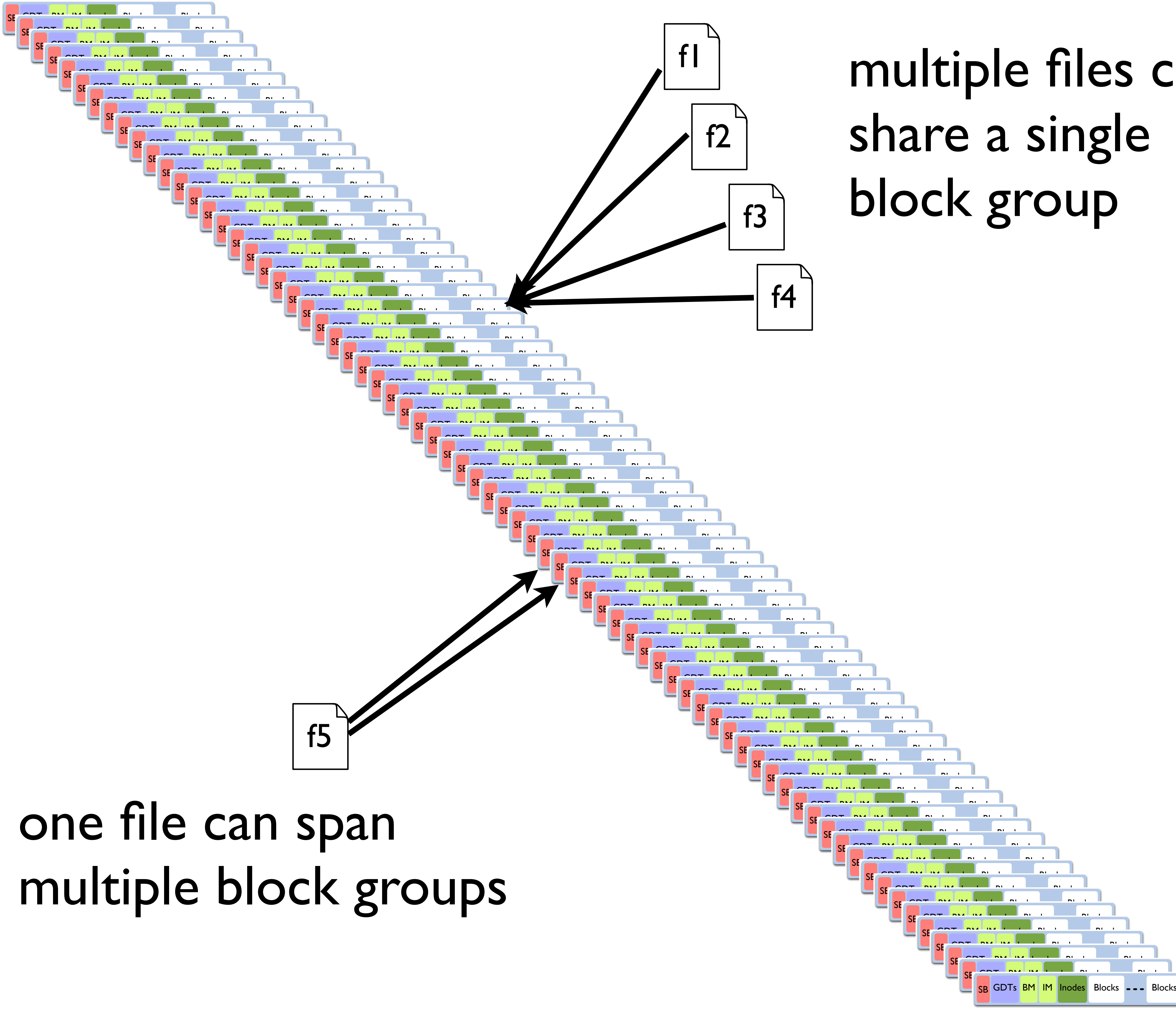
one file can span multiple block groups



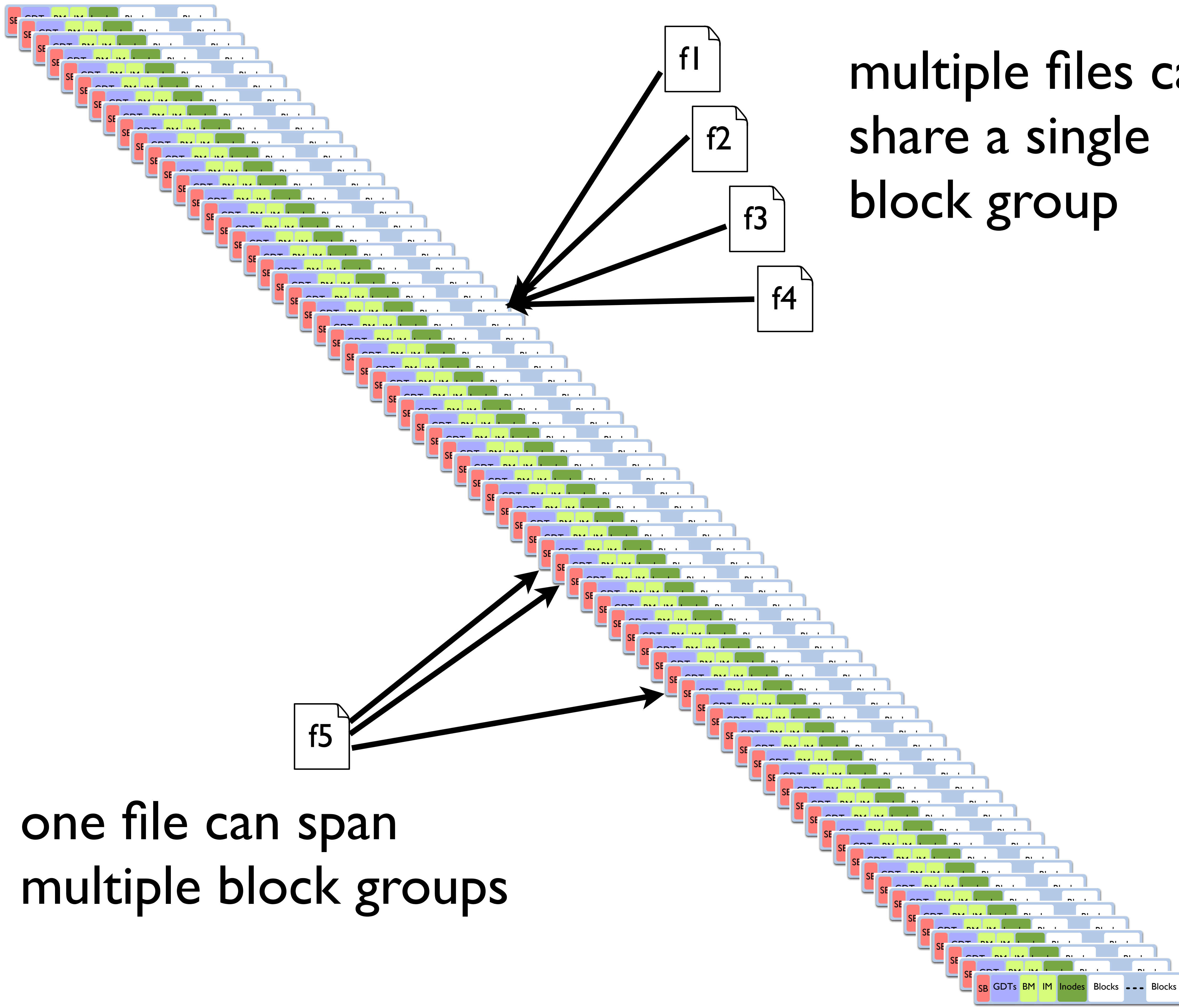
multiple files can share a single block group

one file can span multiple block groups

multiple files can share a single block group

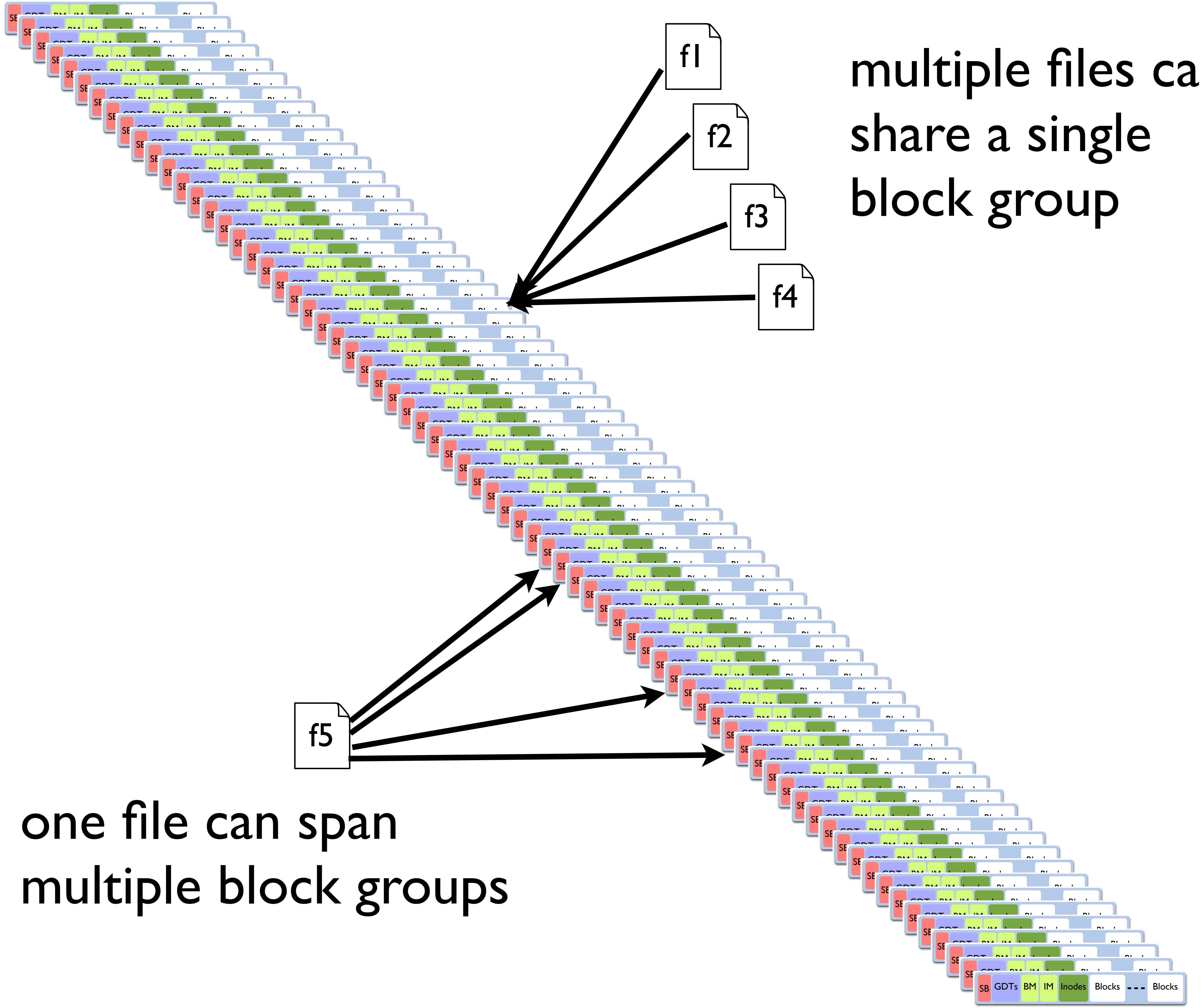


one file can span multiple block groups



multiple files can share a single block group

one file can span multiple block groups



multiple files can share a single block group

one file can span multiple block groups

Layout

Layout

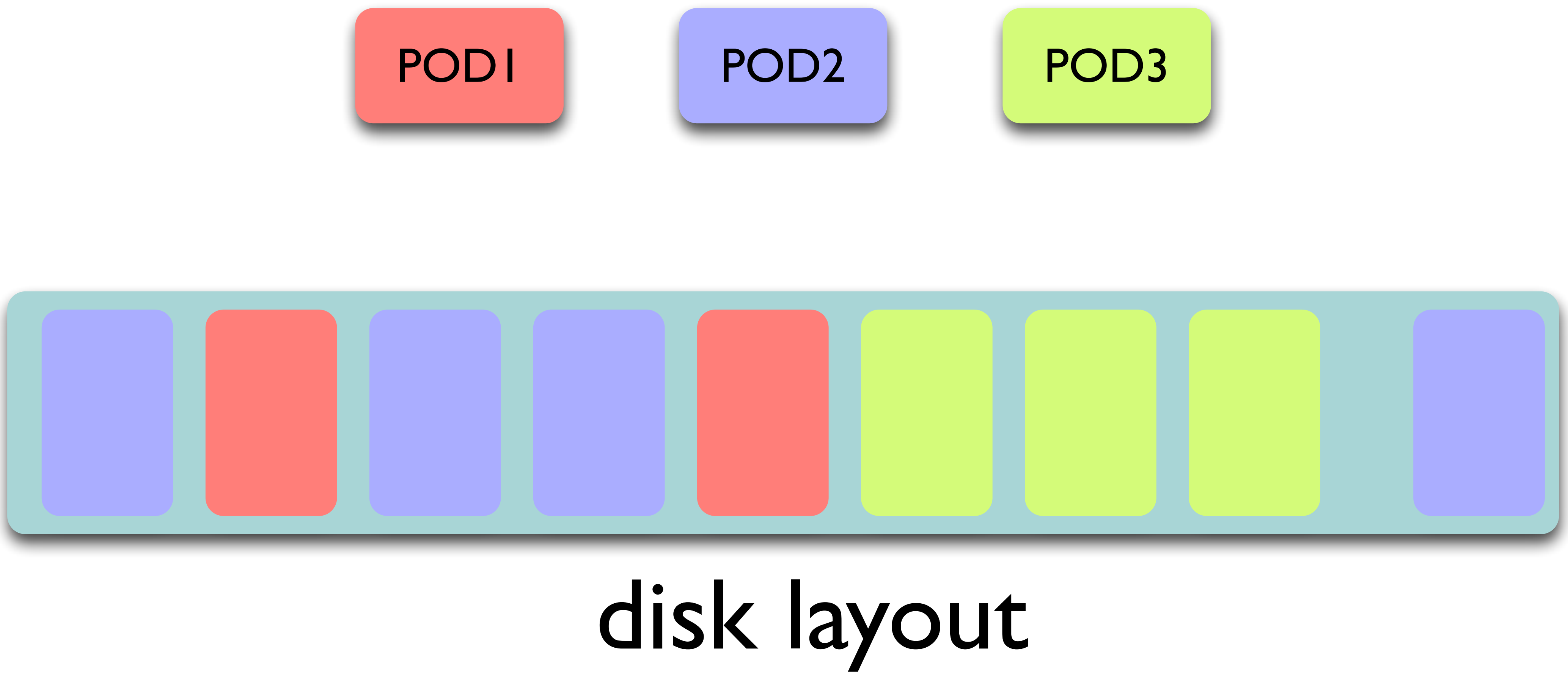
A file pod contains multiple block groups

- one block group only maps to one file pod
- performance locality and fault isolation

Layout

A file pod contains multiple block groups

- one block group only maps to one file pod
- performance locality and fault isolation



Data Structures

Data Structures

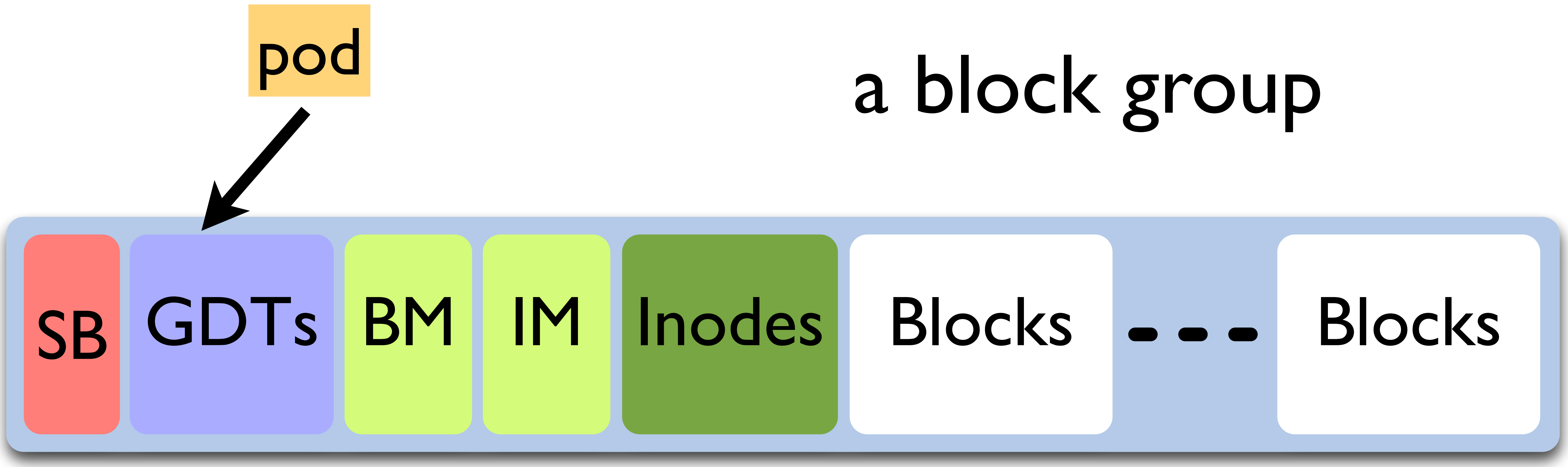
Pod related structure

- ➡ no extra mapping structures

Data Structures

Pod related structure

- no extra mapping structures
- embeds in group descriptors
- group descriptors are loaded into memory

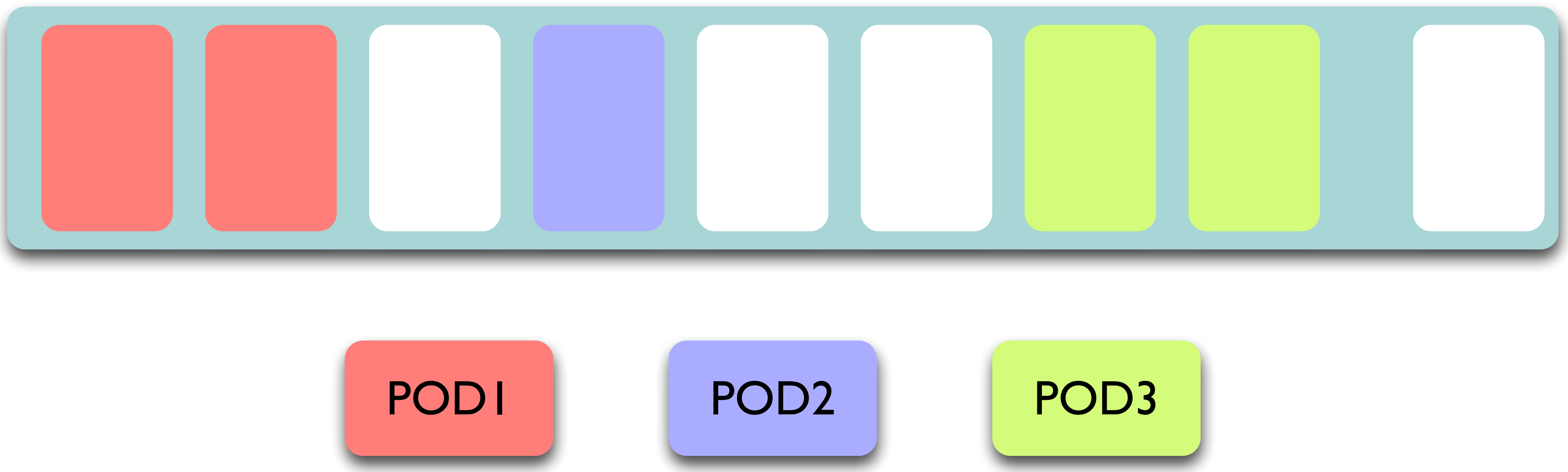


Algorithms

Algorithms

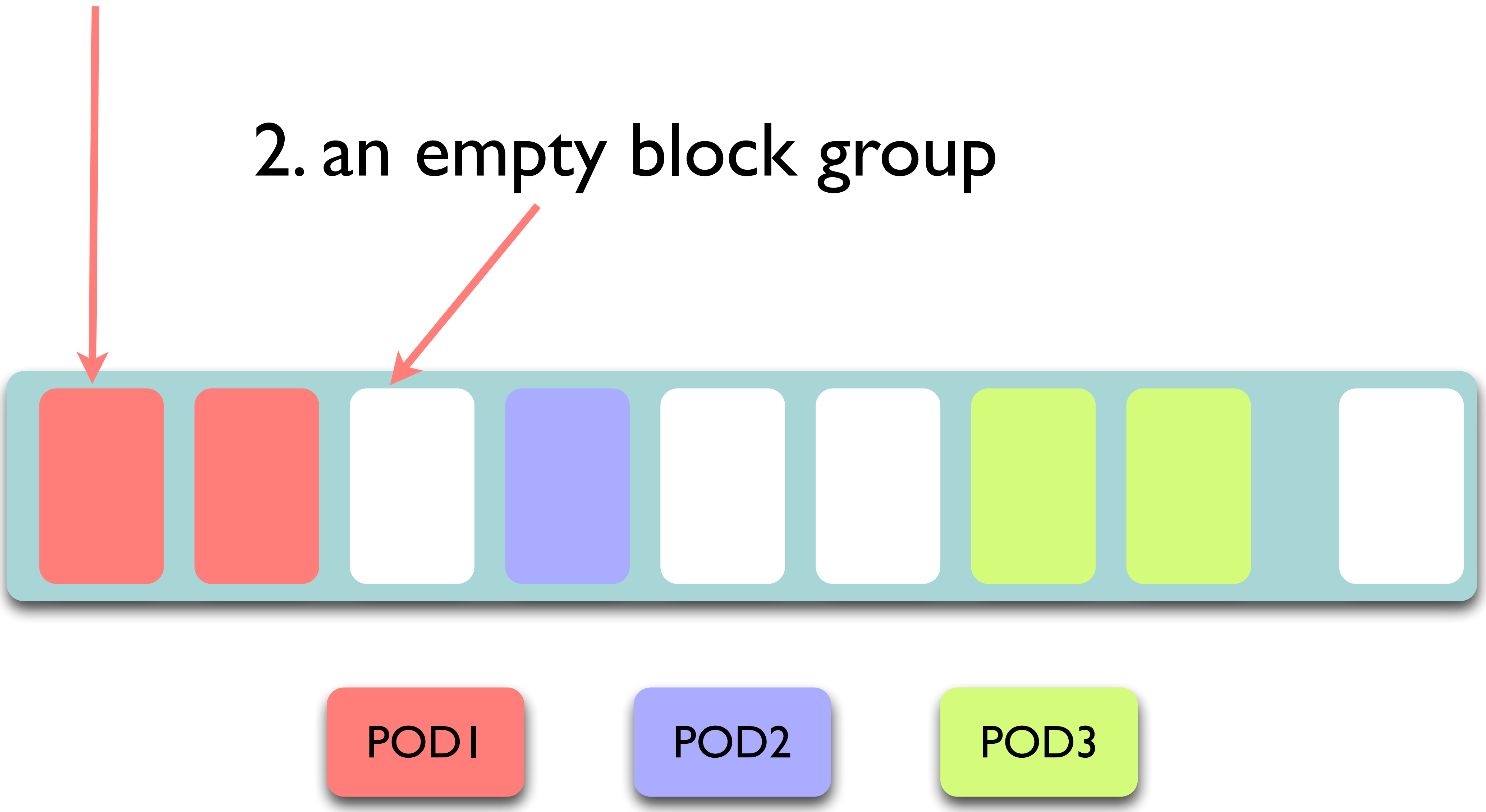
Pod based inode and block allocation

- preserve original allocation's locality
- allocation will not cross pod boundary


POD1
POD2
POD3

1. within the same pod

2. an empty block group



Algorithms

Algorithms

Pod based inode and block allocation

- preserve original allocation's locality
- allocation will not cross pod boundary

De-fragmentation

- potential internal fragmentation

Algorithms

Pod based inode and block allocation

- preserve original allocation's locality
- allocation will not cross pod boundary

De-fragmentation

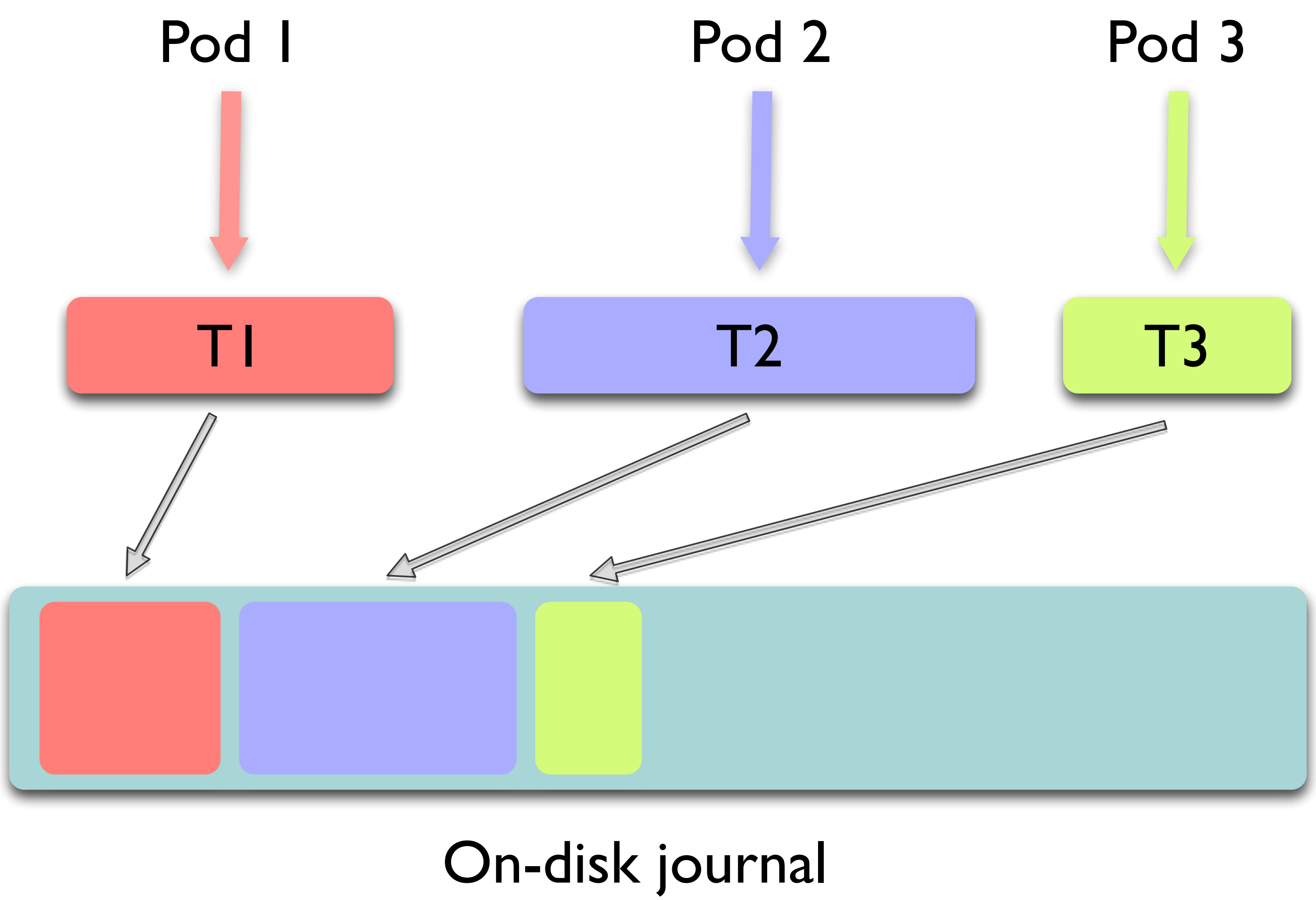
- potential internal fragmentation
- de-fragmentation for file pods
- similar solution in Ext4

Journaling

Journaling

Virtual transaction

- contains updates only from one pod

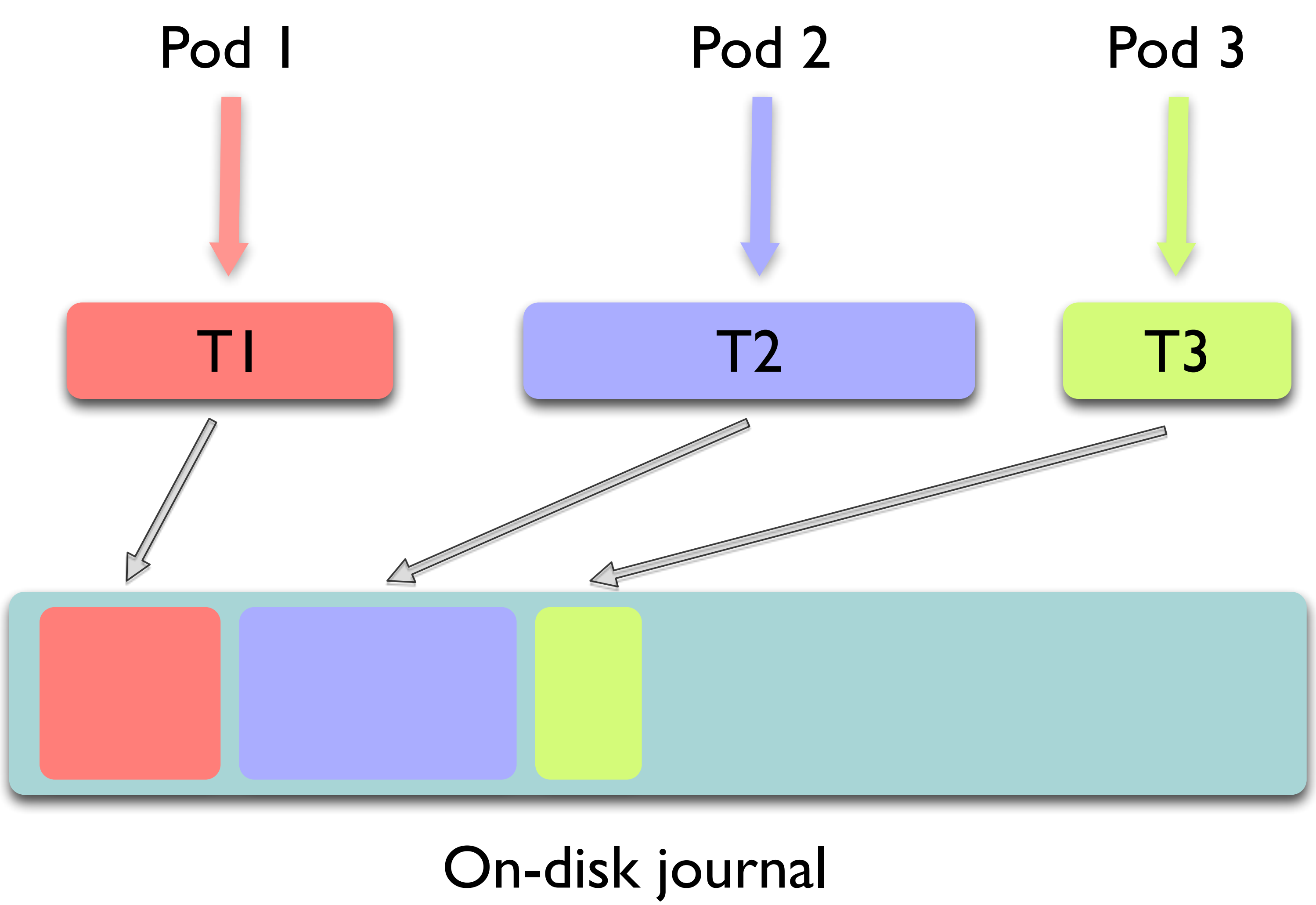


independent transactions

Journaling

Virtual transaction

- contains updates only from one pod
- better performance isolation

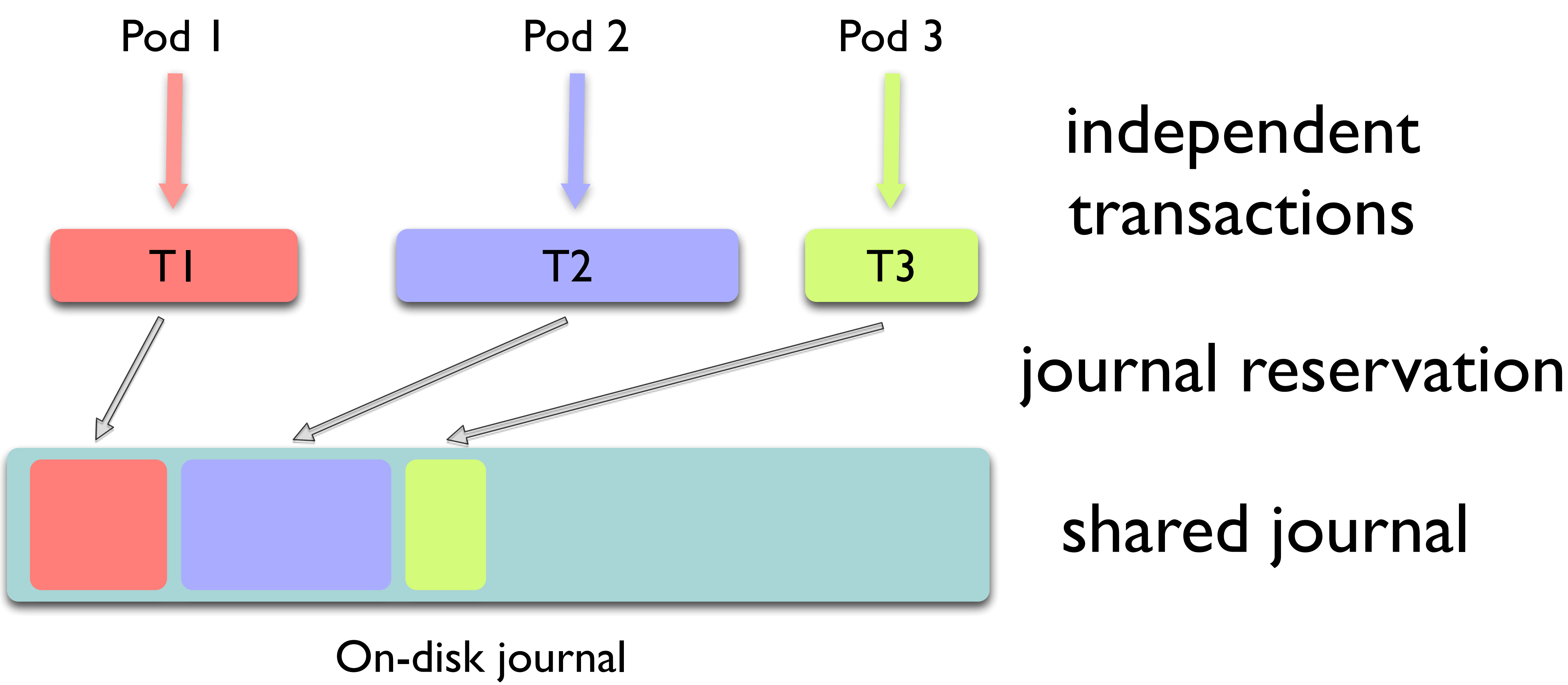


independent transactions

Journaling

Virtual transaction

- contains updates only from one pod
- better performance isolation
- commit multiple virtual transactions in parallel



Introduction

Study of Failure Policies

Isolation File Systems

- New Abstraction
- Fault Isolation
- Quick Recovery
- Preliminary Implementation on Ext3

Challenges

Status

Status

What we did

- a simple prototype for Ext3
- provide readonly isolation

Status

What we did

- a simple prototype for Ext3
- provide readonly isolation

What we plan to do

- crash isolation

Status

What we did

- a simple prototype for Ext3
- provide readonly isolation

What we plan to do

- crash isolation
- quick recovery after failure

Status

What we did

- a simple prototype for Ext3
- provide readonly isolation

What we plan to do

- crash isolation
- quick recovery after failure
- other file systems: Ext4 and Btrfs

Challenges

Challenges

Metadata isolation

- tree-based directory structure
- globally shared metadata: super block, journal
- shared system states: block allocation tree

Challenges

Metadata isolation

- tree-based directory structure
- globally shared metadata: super block, journal
- shared system states: block allocation tree

Local failure

- is it correct to continue to run ?
- light-weight, stateless crash for a pod

Challenges

Metadata isolation

- tree-based directory structure
- globally shared metadata: super block, journal
- shared system states: block allocation tree

Local failure

- is it correct to continue to run ?
- light-weight, stateless crash for a pod

Performance

- potential overhead of managing pods
- better performance isolation
- better scalability

Failure is not an option.

Failure is not an option.

-- NASA

Global failure is not an option;

Global failure is not an option;

local failure with quick recovery

Global failure is not an option;

local failure with quick recovery

is an option.

Global failure is not an option;

local failure with quick recovery

is an option.

-- Isolation File Systems

Questions ?