



Spillover Space: Self-Extending Layouts HLD

- Introduction
- 1. Definitions
- 2. Requirements
- 3. Design Highlights
- 4. Functional Specification
- 5. Logical Specification
- 6. State
- 7. Use Cases
- 8. Analysis
- 9. Deployment
- 10. References

- Introduction
- 1. Definitions
- 2. Requirements
- 3. Design Highlights
- 4. Functional Specification
- 5. Logical Specification
- 6. State
- 7. Use Cases
- 8. Analysis
- 9. Deployment
- 10. References

Title	Spillover Space: Self-Extending Layouts HLD
Summary	...
Jira	 LUS-2528 - Spillover space IN PROGRESS
Owner	Patrick Farrell
Version	1.0
Last Update	 15 Mar 2018

This document presents a High Level Design (HLD) of a partial implementation of spillover space functionality (As described in LUS-2528), specifically self-extending Layouts.

The main purposes of this document are:

- (i) To describe the general problems spillover space is intended to solve
- (ii) To describe the more limited problems solved by the self-extending layouts feature
- (iii) To give a high level overview of the proposed design for self-extending layouts*
 - Depending on length, this section may replace a separate detailed design doc.

The intended audience of this document consists of architects, designers and developers.

Introduction

New storage tech (specifically flash/NVME) is changing the cost/benefit tradeoffs in purchasing file system space and capacity, which is driving an interest in multi-tiered HPC file systems with different storage technologies at each tier. A typical example might be an 'inner' tier on flash (small in size but very high bandwidth and iops), a traditional HDD tier (moderate bandwidth, moderate iops, moderate capacity), and a shingled-magnetic-recording tier (low bandwidth, horrible iops, high capacity). This sort of tiering is more useful if filesystems have functionality to support it. In particular, better data layout functionality, which can help automate choices about where to put data, is extremely helpful in addressing issues with tiering.

One weakness of a tiering implementation is now it's possible to run out of space in one tier, but have plenty in another. It's frustrating for users to get ENOSPC when there's plenty of free space remaining in other parts of the filesystem. Preventing ENOSPC entirely is a very complex problem, reaching in to issues around sparse files, client side dirty data and grants, flushing writes on layout changes, fragmentation as different OSTs run out of space, etc. This proposal does not attempt to address that directly, but instead, it hopes to improve OST allocation and layout policies enough so that ENOSPC can be eliminated in most situations where it is practical to do so (ie, where there is plenty of space elsewhere in the filesystem).

The main thrust of the proposal is to create a file layout with components of not-infinite length, followed by unassigned space (referred to as "extension space") in an uninitialized component (or components). When file access reaches this unassigned space, the system can check if there is still a reasonable (or sufficient - read on for further discussion) amount of space available on the current OSTs. If there is enough space, it will extend the length of the current real component (that preceding the extension space), allowing the file to continue on the same OSTs. If there is not enough space on the current OSTs, it will not extend the current component, and will instead modify the layout to switch to a new component on new OSTs. This can be done by switching to other OSTs within the same set (which could be a pool, probably corresponding to a tier, or the whole file system), or by switching to OSTs selected from a new pool/tier, depending on the specific layout. This does not solve all ENOSPC issues, in particular, files must use this layout type (meaning it does not apply to files manually striped to a different layout or pre-existing files), and the layout must be granted in large "chunks", which means it is possible to run out of space while

using already granted layout. But if applied as a default layout policy, it should both help explicitly avoid ENOSPC issues for file using this layout, and improve OST allocation by increasing allocation granularity & responsiveness. This second aspect will indirectly benefit even files not using this layout type.

1. Definitions

PFL - Progressive File Layouts - A new Lustre layout type introduced in 2.10. Allows multiple layout "components" which allow different stripe layouts for different regions of a file.

FLR - File Level Redundancy/Replication - A layout feature which allows mirrored files by having multiple layout replicas, some of which can be stale.

Component instantiation - Components start out uninstantiated (except for component 0), meaning they do not have specific OST objects assigned. Components are instantiated once a client attempts to write to them. Once a component is instantiated, it can never shrink in size (because it might cover dirty data on a client).

Extension space component/virtual component - A special type of component which is never instantiated, but is used to track space intended for extending the lengths of neighboring components. The central piece of the proposed self-extending layout implementation.

2. Requirements

Allow layout components to grow in size when the OSTs on it meet the extension space policy check (essentially, a check to see if they have "sufficient free space", for various possible definitions of "sufficient free space"). If they do not, attempt to "spill over" to a new layout component to allow the client to continue writing to the file. This new layout component could either be an already specified next component (the obvious use case for this is a component on one tier of tiered storage "spilling over" to a component on the next tier of storage), or if there is not already another component, the system will attempt to create a new component. This new component will use the striping information from the previous component as a template, copying count and/or pool information. This allows a file to 'spillover' even if a particular next component was not specified.

An effective extension space policy for deciding whether or not to extend on a given set of OSTs must be decided on. The simplest policy is just to use the existing OST out of space threshold as used by the stripe allocator, but this policy may need to be replaced with something more capable.

Have a straightforward setstripe command (and C API) for creating these new layouts. Also support the new YAML layout specification format.

Interoperate correctly with file-level redundancy feature and other new layout features, particularly Data on MDT and FLR. (Currently, this appears to be a no-op - The layout types appear to compose cleanly and in a way that does not generate any unusual behavior. There are some mild complexities around FLR.)

3. Design Highlights

The implementation will be almost entirely server side, with the only client side changes being in the userspace tools to allow the client to express and display these layouts.

The key implementation idea is an extension space component or virtual component. This is a component that will never be init. When the client encounters a component which is not init, it sends a layout intent to the server, then restarts i/o and asks for an updated layout. This allows the server to make complex manipulations without directly involving the client. (It is even possible to remove or add layout components at this time, which is something we take advantage of.)

In our case, the manipulation we make is to adjust neighboring "real" components so their extents cover the layout request from the client, rather than initializing the virtual component. The client then proceeds to do i/o to these components, which now cover the required extent of the file.

4. Functional Specification

"What it does"

The functional specification will focus on the core of the feature, which is the code which takes the self-extending PFL layout as described by the user-specified layout and modifies/extends it in response to client access.

When a client attempts to access a portion of the file where the layout is not instantiated (such as an extension space component), it contacts the MDS and informs the MDS of the operation it wants to perform. The MDS must then reply with a layout which permits this operation. In the case where a client access is in an extension space component, the MDS modifies the layout by granting new real layout through one of several possible manipulations described below. The effect is to provide real layout in place of the extension space, permitting I/O to continue. The new layout is granted in a way that is aware of free space/out of space issues on the various OSTs.

This is implemented through the use of the never initialized extension space component (described in detail elsewhere) to grant layout in chunks, then once an imminent out of space condition is detected, spill over to a new layout rather than grant further space in the current layout.

The simplest case is where we have a layout specified to use, for example, two stripes from 'pool1', with a specified length of, say, 10 GiB. This is followed by extension space up to, say, 100 GiB. This is followed by an ordinary component with 4 stripes, on (for example) pool2, which goes from 100 GiB to infinity.

When the file is accessed beyond 10 GB but below 100 GB, an out of space check is performed to verify there's sufficient space on the OSTs used by the existing component. If that check passes, more layout is granted in that first component, which reduces the size of the extension space component.

If that check fails, then we "spill over" to the next component. No more layout is given to the current real component, instead, we delete the extension space component and move the start of the component after the extension space to meet the end of the first component. We have now "spilled over" and i/o will continue to this component, rather than attempt to continue using the first component and potentially run out of space.

It is possible to have more than one extension space component/spillover set up in a given layout, making it possible to spill over between multiple pools, like "pflash, pdisk, pslowdisk", as either the specified extent is exhausted (identically to normal PFL) or out of space checks fail when granting more layout.

There are three additional ways of using this worth highlighting:

1. The first is something called self spillover. This is a replacement for the standard, simple zero to infinity, non-PFL layout, with one component and (necessarily) consistent striping throughout. Instead, we specify a component of some particular length [with a defined striping, possibly w/a pool] and follow it with an extension space component that goes to infinity.

When we go to extend this component and the out of space check fails, there is no further component to spill over to, so instead, we create one. We do this by repeating the existing component, but with new striping. We take the striping of the first component as a template, and ask the stripe allocator to allocate stripes according to that template. This means count, pool, size, etc. are retained, but we will select new OSTs. These may be the same as the old ones, but ideally, these ones will have free space, where the previous ones did not. We perform an out of space check on the new OSTs before actually instantiating this new layout component, and if it fails, we do not create the new component. Instead, we extend the existing component, so as to not return ENOSPC while space remains on those OSTs. We do *not* stripe to new OSTs in the case where they report low on space (even if they have more free space than the existing ones), because if we did, we would end up repeatedly restriping the same files to new OSTs as the system filled up, chasing ever-dropping free space. We would end up with many, many layout components and a very small improvement in space usage.

2. The next detail to highlight is the introduction of zero length components. These are components which have no length (ie start == end) until instantiated. Having them complicates the layout API sanity checking, but they are fundamental to a good implementation of self-extending layouts.

When we switch to a new component, it is important that - if at all possible - we not simply start our new layout on OSTs that are already out of space. So we introduce the idea of zero length uninstantiated components followed by extension space. When one of these components is reached, we ask the stripe allocator to Do the striping operation, and then check the resulting OSTs. If they fail our out of space check, then Instead of extending and using that component, we attempt to spill over to any further specified layout, Where we hope there will be space. If there is no further layout specified, then we will extend the zero length component, in order to allow the user to continue writing until a hard out of space condition is reached, rather than stop them early.

3. The last unusual aspect here is "trailing components", which refers to components which follow an extension space component which goes to infinity.

Imagine a simple self-extending layout, with a normal component with two stripes on pool1 followed by extension space. Now consider that we may not wish to limit the amount of the file that can be placed on OSTs from pool1, but we're still concerned about running out of space in pool1 and would like to be able to spill over to pool2. Because we want to spill over to another pool, the restriping behavior described in '1' won't work.

Instead, we need to be able to specify layout following a layout component which goes to -1.--- Need a clean, tight explanation here. It's coming.

Further diagrams and other details are found in this slide deck:

[LUG DD 2018 - SEPFL.pptx](#)

5. Logical Specification

"How it does it"

The logical decomposition of this feature ends up very simple. The basic logical requirement is that we intercept layout intent operations that would normally initialize a component, when they would attempt to initialize one of our virtual/extension space components. We perform our extension space processing, and the result is a layout that no longer leads to the layout intent touching the virtual component. At that point, we continue on to the normal component initialization processing (which may not result in initializing any components).

This means that all the logic is confined to one layer of Lustre, the LOD (logical object device) layer of the MDS. In fact, all of the logic is implemented in one function (and its children) which handles the layout modifications. This function performs the layout manipulations required to implement the extension space policy, then passes the updated layout back to the higher level intent processing function, which continues on to see if it should instantiate new components.

This means we must insert our extension space handling function anywhere we instantiate objects. This is three places:

`lod_declare_update_plain`

`lod_declare_update_rdonly`

`lod_declare_update_write_pending`

This should be fairly straightforward in all of these places. The extension space handling function guarantees that none of these functions will interact directly with an extension space component. An essential invariant (already alluded to) for this function is that after it executed, the layout is modified such that the layout intent from the client **no longer touches an extension space component**.

There are a few areas of the internal design to highlight. The first is the "out of space policy" implementation. This is the code responsible for using current OST state to determine whether or not to extend the layout on the current OSTs, and if we extend the layout on the current OSTs, **how much** to extend it.

The current design uses a very simple yes/no policy for whether or not to extend on a given OST. Essentially, it asks if the OST has sufficient free space for the extension, if the extension were fully written to. OSTs have a minimum threshold for free space, and once there is less than that available, they report "out of space" to the stripe allocator. The remaining space on the OST is reserved so files already on that OST can still be written to. This threshold is used as the limit when deciding whether or not to extend a layout.

So, for example, for a 100 GB extension granularity with 2 OSTs, each OST is checked to see if it has at least 50 GB (100/2) of available space before it reaches the out of space threshold. If there is sufficient space, the layout is extended on the current OSTs. If there is not sufficient space, we will attempt to move to a new layout component (either through spillover or repetition).

The next details to highlight are component deletion & repetition. Either when an extension space component is exhausted or when we choose not to extend and instantiate a zero length normal component, we must delete that layout component. This is relatively straightforward, and is done by allocating a new layout, which does not have space for the deleted components. Then, we simply copy the components from the old layout which are not deleted, then free the old layout, and use the newly allocated layout. One oddity this introduces is that component IDs are no longer dense - if we have a layout with five components and no mirrors, the components are numbered 1 2 3 4 5. If we delete the third component, obviously the numbering is now 1 2 4 5. This looks a little odd, but does not cause any problems.

Repeating components is closely related to deleting components. It uses the same basic approach of allocating a new layout, except of course larger rather than smaller. It then copies the relevant properties of the component being repeated (specifically, the striping template information) and inserts that component. This has a similar odd effect on component numbering, in that, using our example from above, if 4 is an "infinite length" extension space component & 4 runs out of space on its OSTs, we would create a new component (6), which ends up placed like this:

1 2 3 4 6 5

Again, aside from looking weird, this does not cause any problems.

The first prototype tried cleaning up the component IDs so they had the expected values, but subsequently, it was discovered that some user space code depends on the component IDs for particular components remaining the same.

The last core design component that seems worth highlighting is the behavior of what I refer to as trailing components. These are layout components follow an extension space component of infinite length, and their purpose is described in more detail in the functional specification above.

When such an infinite length extension space component would be deleted (because it failed an out of space check for an extension), if there are no further components, we perform the 'repeat' operation described above. If, however, it is followed by trailing components, we spill over to them instead. Because the trailing components have their start at -1 and their end specifies a "length" rather than an explicit end point (because this would be impossible as we do not know where they will go in the file when they are created), we must modify their start/end points once the -1 component which precedes them (and has the effect of hiding them) is removed. For these trailing components, means replacing the -1/infinity "start" with the end of the preceding component (starting from the first & continuing on), and adding the new "start" of each component to its "end". This places the end in a valid absolute/real location in the layout, by using it as a "length" relative to the start.

5.1. Conformance

The core spillover space requirement about changing layout component extensions to move writing between pools and OSTs to avoid ENOSPC is discharged by the extent manipulations described in the previous sections.

The requirement for a good policy for when to stop granting layout on OSTs may not be fully discharged by this design. Instead, a very simple implementation is described, which will be implemented in the prototype and tested. As long as this proves mostly sufficient, we may

keep it for version 1 - This component is well isolated from the rest of the implementation and could easily be improved on later without affecting the rest of the implementation.

The straightforward setstripe API and YAML interoperability requirements are addressed by simple changes to the userspace tools. These changes are simple enough that they are not described in detail.

The last requirement - to interoperate correctly with other layout features - is met mostly naturally by this design. We insert our processing logic in to other layout operations so that no other layout operations ever interact directly with our virtual components. This means that no further changes to those operations/features should be required.

5.2. Dependencies

This implementation is constrained entirely to the LOD layer on the server and the lfs/llapi code on the client. The current design introduces minimal dependencies beyond those layers.

5.3. Security

This feature should have no security implications. All of its operations are contained within "setstripe" and normal file i/o operations, which already have existing security models which are not affected by this.

6. State

7. Use Cases

There are two primary use cases.

1. A system with multiple tiers of OSTs. This is, in essence, a hardware requirement to have different OST types and the desire to "spill over" from one set to another. The classic example is a tier of small, fast OSTs on flash, and then a second tier of larger, slower OSTs on spinning disk. Files would be created on flash and data would be written their either until the file reached an upper size limit or the OST(s) ran low on space. Then the layout would automatically "spill over" to the second tier of larger, slower OSTs on spinning disk. This should help users avoid ENOSPC errors when a particular OST or set OSTs fills up and there is still space free in another OST pool.

This gives more flexibility than a simple progressive file layout, such as 0-100 GB on flash, then 100 GB to infinity on disk. If flash runs out of space before 100GB, then writes will fail. A SEPFL layout might be 0-10GB, extending 10 GB at a time, upper limit of 200 GB, then to infinity on disk. If the flash tier gets low on space, the layout will automatically "spill over" to the disk tier.

2. When there is only one set of OSTs, the feature can still be used to help avoid ENOSPC. In this case, a self-extending layout can be created that only uses one tier. Something like: 0-100 GB, extending 100 GB, upper limit infinity. When a write occurs after 100 GiB, the layout will be extended on the current OSTs, if there is enough space. If there is not enough space, this will use the "self-spillover"/repeated component behavior of SEPFL. Another component is created using the striping information from the first component as a template, but with new striping requested from the stripe allocator. If there is sufficient free space on this component, the layout is extended on this component. This means that if one OST starts running out of space, the layout can adjust to use different OSTs, even in absence of explicit tiers.

As above, this would be a layout template that would be applied to files by default. This should reduce the impact of running low on space on a particular OST.

7.1 Scenarios

7.2 Failures

This feature does not introduce any obvious new failure-related issues - It uses the existing infrastructure for handling failure in the client-server connection (replay) and on the server (failover, fsck).

8. Analysis

8.1 Scalability

This feature is expected to have two scalability related concerns.

1. A larger number of layout components increases layout size, taking up more space on the MDT and requiring more time to send to the client. This is a problem common to other layout enhancements and to PFL itself. As the number of layout components rises, these requirements increase with it. There is no specific solution to this, but it is mitigated by not creating overly complex layouts. This problem is not expected to be serious as this feature will only occasionally create **new** layout components and otherwise has a marginal impact on the number of layout components vs a traditional PFL layout.
2. When client i/o reaches an extension space component, we must contact the MDT for additional layout in order to continue our I/O. This creates a delay on the client and work on the MDT. If clients must contact the MDT too often, the burden could become significant. The solution is simply to make the extension size (the amount of new layout granted) relatively large. There is a tradeoff - Larger extension sizes mean we must contact the metadata server less, but the granularity of space allocation is increased. This makes it more likely we will run out of space. Tuning this will be an important part of end-user configuration.

Both of these issues should be addressable by appropriate configuration.

8.2 Limitations

This section is intended to describe limitations to the expected functionality of the feature versus the design. This is NOT intended to describe intrinsic limitations of the design, rather it is intended to describe areas where the realized functionality falls short of the design due to outside factors.

There are two principal limitations that apply to this currently, both are limitations for progressive file layout files and are inherited here.

1. Appending to PFL files instantiates the entire layout. (LU-9341 at Intel) Currently, when a file is written to with O_APPEND, the entire layout must be instantiated. This is because knowing the current file size (required for append) requires locking all data components (so the file cannot be written to at a higher offset), and it is difficult to ensure no new data components will be created in a PFL file with uninstantiated components. The simple solution is to instantiate the entire layout.

That creates a write intent layout request from 0 to EOF. The current impact on SEPFL files is to extend all normal components to their upper limit and (as a result) remove all virtual/extension space components. There is a proposed fix in LU-9341, but no code to implement it yet.

2. Group locks have a similar issue, but the solution is distinct. Since a group lock is supposed to cover all of a file and be exclusive, it must lock all data components. As with appending, this requires fully instantiating the file. This has the same impact on SEPFL files as appending - It extends all components to their upper limit and virtual/extension space components. This is tracked in LU-9479. There is a proposal for a fix (essentially, group locks as layout locks), but it's just a rough idea and there is no code to implement it.

8.3 Testing

The implementation will include a number of Lustre sanity tests, exploring the basic functionality. These should cover basic functional testing, but there is also a requirement for system test to do scale testing, etc.

Some ideas for focus areas in system test include:

Many clients doing varied I/O to a file with a self-extending PFL layout (shared file writing, the write_disjoint test from the Lustre sanity suite,

Testing with multiple pools of OSTs, filling one pool up and verifying spill over to the second works as expected

Testing of layout templates/default layouts, verifying inheritance and correct behavior (There will be at least one sanity test for this as well)

Testing of spillover behavior with multiple files & pools (Example: Create a SEPFL template/default layout to spill over from one pool to another, then write to multiple files until the first pool is low on space, and verify they spill over to the second pool successfully)

Generalized test runs with a self-extending PFL layout template set as the default layout. IE make all files without explicit striping in to SEPFL files, and run various tests.

Failover/failback testing while writing to self-extending PFL files. Verify no unexpected behavior.

8.4 Rationale

There are two main design elements with obvious alternatives.

1. The use of "virtual" components for extension space. We could instead have changed the behavior of space which had no layout defined at all. This would simplify things a bit conceptually for users - Virtual components are a bit odd. They really want to describe the behavior of the real component, and thinking about a virtual component is an extra burden which could be removed.

However, the virtual and uninitialized component has a number of advantages. It allows us to retain the existing behavior with uninitialized layouts, where a layout intent is sent to the server, the layout is refreshed, and i/o is restarted. This means clients that do not know about self-extending layouts can nevertheless read and write to them correctly. The virtual extension space component also gives us a place to store the extension space properties, such as its extent and the extension size. This allows us to avoid adding fields to the standard layout component, which would be unused for most components and waste storage space. This also lets us keep the on disk format the same, which eases version compatibility issues.

2. When extending the layout, we could either increase the dimensions of an existing component, or create a new layout component to cover the new space. The second option (which was the original design proposed in conversations at LAD) has a few advantages. In particular, it would be possible to address each component separately, for example with HSM release/restore. (This would require other changes, but it would be possible.) This is appealing, but it has a basic flaw that seems to make it impractical. Every time we add a layout component, it consumes space on the MDT and increases the layout size, which is eventually a performance problem. This means we want to keep the extension size relatively high, so we do not have to add too many components. But this is exactly the opposite of what we want if our goal is to use the components for partial release/restore. For that, we would like to be able to address relatively small pieces of the file, which means many components, which is highly impractical. Because we cannot satisfy the HSM case anyway, we chose the approach of increasing the dimensions of an existing component, at least where that's possible.

In the cases where extending the existing component would result in the client running out of space, we **do** attempt to create a new component (with new stripes) to allow the client to continue writing. But this is intended to be a rare situation, and should not create many components.

9. Deployment

Deployment will consist first of installing the required Lustre versions on client and server, and then using the feature. Actual use of the feature will require users/administrators to create layouts which use it. The intention is that default layouts making use of this feature will be created. The details of suggested layouts, etc, will be determined and documentation will be created to associate with the feature.

9.1 Compatibility

The current implementation **should** be mostly compatible with any client which understands progressive file layouts. There are currently **no** client implementation changes planned in Lustre itself to support this functionality. That may change as the implementation progresses, but that is the current state.

It is only "mostly" compatible because a client without this feature will not be able to correctly display self extending layouts. It will not recognize the extension flag or the special length descriptions. This does not impair correct operation because only the server uses these directly, but it could cause confusion.

Clients without this support would also be unable to create SEPFL layouts, since their lfs will not be able to express such a layout.

It may be best to prevent clients that do not understand SEPFL layouts from opening them, as is done with FLR files, but it may not be strictly necessary.

Server side compatibility is more of a problem. If we downgrade from a server with this support to one without it, the server will not handle the layout correctly. So we should make sure to use whatever compatibility mechanism exists for new types of layouts to ensure the server recognizes this and avoids trying to use them.

9.2 Installation

The component will be part of a new Lustre software package, which would be distributed in a NEO update. There are no special installation requirements.

10. References

FLR HLD:

http://wiki.lustre.org/File_Level_Replication_High_Level_Design

Layout Enhancement design:

http://wiki.lustre.org/Layout_Enhancement_High_Level_Design

PFL Design:

http://wiki.lustre.org/Progressive_File_Layouts


LU-10070: PFL self-extending file layout

<https://jira.hpdd.intel.com/browse/LU-10070>

LUS-2528: Spillover Space

<p>LUS-2528 - Spillover space</p> <p>IN PROGRESS</p>
--

- Introduction
- 1. Definitions
- 2. Requirements
- 3. Design Highlights
- 4. Functional Specification
- 5. Logical Specification
- 6. State
- 7. Use Cases
- 8. Analysis
- 9. Deployment
- 10. References

Title	Spillover Space: Self-Extending Layouts HLD	
Summary	...	
Jira	<table border="1"> <tr> <td> <p>LUS-2528 - Spillover space</p> <p>IN PROGRESS</p> </td> </tr> </table>	<p>LUS-2528 - Spillover space</p> <p>IN PROGRESS</p>
<p>LUS-2528 - Spillover space</p> <p>IN PROGRESS</p>		
Owner	Patrick Farrell	
Version	1.0	
Last Update	 15 Mar 2018	

This document presents a High Level Design (HLD) of a partial implementation of spillover space functionality (As described in LUS-2528), specifically self-extending Layouts.

The main purposes of this document are:

- (i) To describe the general problems spillover space is intended to solve
- (ii) To describe the more limited problems solved by the self-extending layouts feature
- (iii) To give a high level overview of the proposed design for self-extending layouts*
 - Depending on length, this section may replace a separate detailed design doc.

The intended audience of this document consists of architects, designers and developers.

Introduction

New storage tech (specifically flash/NVME) is changing the cost/benefit tradeoffs in purchasing file system space and capacity, which is driving an interest in multi-tiered HPC file systems with different storage technologies at each tier. A typical example might be an 'inner' tier on flash (small in size but very high bandwidth and iops), a traditional HDD tier (moderate bandwidth, moderate iops, moderate capacity), and a shingled-magnetic-recording tier (low bandwidth, horrible iops, high capacity). This sort of tiering is more useful if filesystems have

functionality to support it. In particular, better data layout functionality, which can help automate choices about where to put data, is extremely helpful in addressing issues with tiering.

One weakness of a tiering implementation is now it's possible to run out of space in one tier, but have plenty in another. It's frustrating for users to get ENOSPC when there's plenty of free space remaining in other parts of the filesystem. Preventing ENOSPC entirely is a very complex problem, reaching in to issues around sparse files, client side dirty data and grants, flushing writes on layout changes, fragmentation as different OSTs run out of space, etc. This proposal does not attempt to address that directly, but instead, it hopes to improve OST allocation and layout policies enough so that ENOSPC can be eliminated in most situations where it is practical to do so (ie, where there is plenty of space elsewhere in the filesystem).

The main thrust of the proposal is to create a file layout with components of not-infinite length, followed by unassigned space (referred to as "extension space") in an uninitialized component (or components). When file access reaches this unassigned space, the system can check if there is still a reasonable (or sufficient - read on for further discussion) amount of space available on the current OSTs. If there is enough space, it will extend the length of the current real component (that preceding the extension space), allowing the file to continue on the same OSTs. If there is not enough space on the current OSTs, it will not extend the current component, and will instead modify the layout to switch to a new component on new OSTs. This can be done by switching to other OSTs within the same set (which could be a pool, probably corresponding to a tier, or the whole file system), or by switching to OSTs selected from a new pool/tier, depending on the specific layout. This does not solve all ENOSPC issues, in particular, files must use this layout type (meaning it does not apply to files manually striped to a different layout or pre-existing files), and the layout must be granted in large "chunks", which means it is possible to run out of space while using already granted layout. But if applied as a default layout policy, it should both help explicitly avoid ENOSPC issues for file using this layout, and improve OST allocation by increasing allocation granularity & responsiveness. This second aspect will indirectly benefit even files not using this layout type.

1. Definitions

PFL - Progressive File Layouts - A new Lustre layout type introduced in 2.10. Allows multiple layout "components" which allow different stripe layouts for different regions of a file.

FLR - File Level Redundancy/Replication - A layout feature which allows mirrored files by having multiple layout replicas, some of which can be stale.

Component instantiation - Components start out uninstantiated (except for component 0), meaning they do not have specific OST objects assigned. Components are instantiated once a client attempts to write to them. Once a component is instantiated, it can never shrink in size (because it might cover dirty data on a client).

Extension space component/virtual component - A special type of component which is never instantiated, but is used to track space intended for extending the lengths of neighboring components. The central piece of the proposed self-extending layout implementation.

2. Requirements

Allow layout components to grow in size when the OSTs on it meet the extension space policy check (essentially, a check to see if they have "sufficient free space", for various possible definitions of "sufficient free space"). If they do not, attempt to "spill over" to a new layout component to allow the client to continue writing to the file. This new layout component could either be an already specified next component (the obvious use case for this is a component on one tier of tiered storage "spilling over" to a component on the next tier of storage), or if there is not already another component, the system will attempt to create a new component. This new component will use the striping information from the previous component as a template, copying count and/or pool information. This allows a file to 'spillover' even if a particular next component was not specified.

An effective extension space policy for deciding whether or not to extend on a given set of OSTs must be decided on. The simplest policy is just to use the existing OST out of space threshold as used by the stripe allocator, but this policy may need to be replaced with something more capable.

Have a straightforward setstripe command (and C API) for creating these new layouts. Also support the new YAML layout specification format.

Interoperate correctly with file-level redundancy feature and other new layout features, particularly Data on MDT and FLR. (Currently, this appears to be a no-op - The layout types appear to compose cleanly and in a way that does not generate any unusual behavior. There are some mild complexities around FLR.)

3. Design Highlights

The implementation will be almost entirely server side, with the only client side changes being in the userspace tools to allow the client to express and display these layouts.

The key implementation idea is an extension space component or virtual component. This is a component that will never be init. When the client encounters a component which is not init, it sends a layout intent to the server, then restarts i/o and asks for an updated layout. This allows the server to make complex manipulations without directly involving the client. (It is even possible to remove or add layout components at this time, which is something we take advantage of.)

In our case, the manipulation we make is to adjust neighboring "real" components so their extents cover the layout request from the client, rather than initializing the virtual component. The client then proceeds to do i/o to these components, which now cover the required extent of the file.

4. Functional Specification

"What it does"

The functional specification will focus on the core of the feature, which is the code which takes the self-extending PFL layout as described by the user-specified layout and modifies/extends it in response to client access.

When a client attempts to access a portion of the file where the layout is not instantiated (such as an extension space component), it contacts the MDS and informs the MDS of the operation it wants to perform. The MDS must then reply with a layout which permits this operation. In the case where a client access is in an extension space component, the MDS modifies the layout by granting new real layout through one of several possible manipulations described below. The effect is to provide real layout in place of the extension space, permitting I/O to continue. The new layout is granted in a way that is aware of free space/out of space issues on the various OSTs.

This is implemented through the use of the never initialized extension space component (described in detail elsewhere) to grant layout in chunks, then once an imminent out of space condition is detected, spill over to a new layout rather than grant further space in the current layout.

The simplest case is where we have a layout specified to use, for example, two stripes from 'pool1', with a specified length of, say, 10 GiB. This is followed by extension space up to, say, 100 GiB. This is followed by an ordinary component with 4 stripes, on (for example) pool2, which goes from 100 GiB to infinity.

When the file is accessed beyond 10 GB but below 100 GB, an out of space check is performed to verify there's sufficient space on the OSTs used by the existing component. If that check passes, more layout is granted in that first component, which reduces the size of the extension space component.

If that check fails, then we "spill over" to the next component. No more layout is given to the current real component, instead, we delete the extension space component and move the start of the component after the extension space to meet the end of the first component. We have now "spilled over" and i/o will continue to this component, rather than attempt to continue using the first component and potentially run out of space.

It is possible to have more than one extension space component/spillover set up in a given layout, making it possible to spill over between multiple pools, like "pflash, pdisk, pslowdisk", as either the specified extent is exhausted (identically to normal PFL) or out of space checks fail when granting more layout.

There are three additional ways of using this worth highlighting:

1. The first is something called self spillover. This is a replacement for the standard, simple zero to infinity, non-PFL layout, with one component and (necessarily) consistent striping throughout. Instead, we specify a component of some particular length [with a defined striping, possibly w/a pool] and follow it with an extension space component that goes to infinity.

When we go to extend this component and the out of space check fails, there is no further component to spill over to, so instead, we create one. We do this by repeating the existing component, but with new striping. We take the striping of the first component as a template, and ask the stripe allocator to allocate stripes according to that template. This means count, pool, size, etc. are retained, but we will select new OSTs. These may be the same as the old ones, but ideally, these ones will have free space, where the previous ones did not. We perform an out of space check on the new OSTs before actually instantiating this new layout component, and if it fails, we do not create the new component. Instead, we extend the existing component, so as to not return ENOSPC while space remains on those OSTs. We do *not* stripe to new OSTs in the case where they report low on space (even if they have more free space than the existing ones), because if we did, we would end up repeatedly restriping the same files to new OSTs as the system filled up, chasing ever-dropping free space. We would end up with many, many layout components and a very small improvement in space usage.

2. The next detail to highlight is the introduction of zero length components. These are components which have no length (ie start == end) until instantiated. Having them complicates the layout API sanity checking, but they are fundamental to a good implementation of self-extending layouts.

When we switch to a new component, it is important that - if at all possible - we not simply start our new layout on OSTs that are already out of space. So we introduce the idea of zero length uninstantiated components followed by extension space. When one of these components is reached, we ask the stripe allocator to do the striping operation, and then check the resulting OSTs. If they fail our out of space check, then instead of extending and using that component, we attempt to spill over to any further specified layout, where we hope there will be space. If there is no further layout specified, then we will extend the zero length component, in order to allow the user to continue writing until a hard out of space condition is reached, rather than stop them early.

3. The last unusual aspect here is "trailing components", which refers to components which follow an extension space component which goes to infinity.

Imagine a simple self-extending layout, with a normal component with two stripes on pool1 followed by extension space. Now consider that we may not wish to limit the amount of the file that can be placed on OSTs from pool1, but we're still concerned about running out of space in pool1 and would like to be able to spill over to pool2. Because we want to spill over to another pool, the restriping behavior described in '1' won't work.

Instead, we need to be able to specify layout following a layout component which goes to -1.

The goal of this feature is to use layout modification to avoid encountering out of space as much as possible.

To this end, we make it possible to stop using a particular layout component when one of the OSTs in it is running low on space. We switch instead to a new component, which will be striped to different OSTs, possibly in another pool.

This is implemented through the use of the never initialized extension space component (described in detail elsewhere) to grant layout in chunks, then once an imminent out of space condition is detected, spill over to a new layout rather than grant further space in the current layout.

The simplest case is where we have a layout specified to use, for example, two stripes from 'pool1', with a specified length of, say, 10 GiB. This is followed by extension space up to, say, 100 GiB. This is followed by an ordinary component with 4 stripes, on (for example) pool2, which goes from 100 GiB to infinity.

When the file is accessed beyond 10 GB but below 100 GB, an out of space check is performed to verify there's sufficient space on the OSTs used by the existing component. If that check passes, more layout is granted in that first component, which reduces the size of the extension space component.

If that check fails, then we "spill over" to the next component. No more layout is given to the current real component, instead, we delete the extension space component and move the start of the component after the extension space to meet the end of the first component. We have now "spilled over" and i/o will continue to this component, rather than attempt to continue using the first component and potentially run out of space.

It is possible to have more than one extension space component/spillover set up in a given layout, making it possible to spill over between multiple pools, like "pflash, pdisk, pslowdisk", as either the specified extent is exhausted (identically to normal PFL) or out of space checks fail when granting more layout.

There are three additional ways of using this worth highlighting:

1. The first is something called self spillover. This is a replacement for the standard, simple zero to infinity, non-PFL layout, with one component and (necessarily) consistent striping throughout. Instead, we specify a component of some particular length [with a defined striping, possibly w/a pool] and follow it with an extension space component that goes to infinity.

When we go to extend this component and the out of space check fails, there is no further component to spill over to, so instead, we create one. We do this by repeating the existing component, but with new striping. We take the striping of the first component as a template, and ask the stripe allocator to allocate stripes according to that template. This means count, pool, size, etc. are retained, but we will select new OSTs. These may be the same as the old ones, but ideally, these ones will have free space, where the previous ones did not. We perform an out of space check on the new OSTs before actually instantiating this new layout component, and if it fails, we do not create the new component. Instead, we extend the existing component, so as to not return ENOSPC while space remains on those OSTs. We do *not* stripe to new OSTs in the case where they report low on space (even if they have more free space than the existing ones), because if we did, we would end up repeatedly restriping the same files to new OSTs as the system filled up, chasing ever-dropping free space. We would end up with many, many layout components and a very small improvement in space usage.

2. The next detail to highlight is the introduction of zero length components. These are components which have no length (ie start == end) until instantiated. Having them complicates the layout API sanity checking, but they are fundamental to a good implementation of self-extending layouts.

When we switch to a new component, it is important that - if at all possible - we not simply start our new layout on OSTs that are already out of space. So we introduce the idea of zero length uninstantiated components followed by extension space. When one of these components is reached, we ask the stripe allocator to Do the striping operation, and then check the resulting OSTs. If they fail our out of space check, then Instead of extending and using that component, we attempt to spill over to any further specified layout, Where we hope there will be space. If there is no further layout specified, then we will extend the zero length component, in order to allow the user to continue writing until a hard out of space condition is reached, rather than stop them early.

3. The last unusual aspect here is "trailing components", which refers to components which follow an extension space component which goes to infinity.

Imagine a simple self-extending layout, with a normal component with two stripes on pool1 followed by extension space. Now consider that we may not wish to limit the amount of the file that can be placed on OSTs from pool1, but we're still concerned about running out of space in pool1 and would like to be able to spill over to pool2. Because we want to spill over to another pool, the restriping behavior described in '1' won't work.

Instead, we need to be able to specify layout following a layout component which goes to -1. This is done with what we refer to as trailing components. These components are unusual in that they are all specified as having their start at -1/infinity. This is required because their true starting point is not known when they are created, and is instead determined by the end point of the preceding component when the system decides to not grant any more layout.

So the endpoints of these components specifies their *length*, rather than an explicit endpoint in the file extent map. When the infinite extension space component preceding them is removed, their start and end points are adjusted to line up with the end of the last instantiated component. As described above, this allows a layout component with no upper bound on the amount of space it can potentially extend to

be followed by a layout which encodes further tiering information, rather than having that infinity/-1 component necessarily be the end of the layout.

Some diagrams and other details are found in this slide deck:

[LUG DD 2018 - SEPFL.pptx](#)

5. Logical Specification

"How it does it"

The logical decomposition of this feature ends up very simple. The basic logical requirement is that we intercept layout intent operations that would normally initialize a component, when they would attempt to initialize one of our virtual/extension space components. We perform our extension space processing, and the result is a layout that no longer leads to the layout intent touching the virtual component. At that point, we continue on to the normal component initialization processing (which may not result in initializing any components).

This means that all the logic is confined to one layer of Lustre, the LOD (logical object device) layer of the MDS. In fact, all of the logic is implemented in one function (and its children) which handles the layout modifications. This function performs the layout manipulations required to implement the extension space policy, then passes the updated layout back to the higher level intent processing function, which continues on to see if it should instantiate new components.

This means we must insert our extension space handling function anywhere we instantiate objects. This is three places:

`lod_declare_update_plain`

`lod_declare_update_rdonly`

`lod_declare_update_write_pending`

This should be fairly straightforward in all of these places. The extension space handling function guarantees that none of these functions will interact directly with an extension space component. An essential invariant (already alluded to) for this function is that after it executed, the layout is modified such that the layout intent from the client **no longer touches an extension space component**.

There are a few areas of the internal design to highlight. The first is the "out of space policy" implementation. This is the code responsible for using current OST state to determine whether or not to extend the layout on the current OSTs, and if we extend the layout on the current OSTs, **how much** to extend it.

The current design uses a very simple yes/no policy for whether or not to extend on a given OST. Essentially, it asks if the OST has sufficient free space for the extension, if the extension were fully written to. OSTs have a minimum threshold for free space, and once there is less than that available, they report "out of space" to the stripe allocator. The remaining space on the OST is reserved so files already on that OST can still be written to. This threshold is used as the limit when deciding whether or not to extend a layout.

So, for example, for a 100 GB extension granularity with 2 OSTs, each OST is checked to see if it has at least 50 GB (100/2) of available space before it reaches the out of space threshold. If there is sufficient space, the layout is extended on the current OSTs. If there is not sufficient space, we will attempt to move to a new layout component (either through spillover or repetition).

The next details to highlight are component deletion & repetition. Either when an extension space component is exhausted or when we choose not to extend and instantiate a zero length normal component, we must delete that layout component. This is relatively straightforward, and is done by allocating a new layout, which does not have space for the deleted components. Then, we simply copy the components from the old layout which are not deleted, then free the old layout, and use the newly allocated layout. One oddity this introduces is that component IDs are no longer dense - if we have a layout with five components and no mirrors, the components are numbered 1 2 3 4 5. If we delete the third component, obviously the numbering is now 1 2 4 5. This looks a little odd, but does not cause any problems.

Repeating components is closely related to deleting components. It uses the same basic approach of allocating a new layout, except of course larger rather than smaller. It then copies the relevant properties of the component being repeated (specifically, the striping template information) and inserts that component. This has a similar odd effect on component numbering, in that, using our example from above, if 4 is an "infinite length" extension space component & 4 runs out of space on its OSTs, we would create a new component (6), which ends up placed like this:

1 2 3 4 6 5

Again, aside from looking weird, this does not cause any problems.

The first prototype tried cleaning up the component IDs so they had the expected values, but subsequently, it was discovered that some user space code depends on the component IDs for particular components remaining the same.

The last core design component that seems worth highlighting is the behavior what I refer to as trailing components. These are layout components follow an extension space component of infinite length, and are described in more detail in the functional specification above.

When such an infinite length extension space component is deleted (because it failed an out of space check for an extension), we must then spill over to the somewhat unusual trailing components.

--- To be continued ^^^^ ---

5.1. Conformance

The core spillover space requirement about changing layout component extensions to move writing between pools and OSTs to avoid ENOSPC is discharged by the extent manipulations described in the previous sections.

The requirement for a good policy for when to stop granting layout on OSTs may not be fully discharged by this design. Instead, a very simple implementation is described, which will be implemented in the prototype and tested. As long as this proves mostly sufficient, we may keep it for version 1 - This component is well isolated from the rest of the implementation and could easily be improved on later without affecting the rest of the implementation.

The straightforward setstripe API and YAML interoperability requirements are addressed by simple changes to the userspace tools. These changes are simple enough that they are not described in detail.

The last requirement - to interoperate correctly with other layout features - is met mostly naturally by this design. We insert our processing logic in to other layout operations so that no other layout operations ever interact directly with our virtual components. This means that no further changes to those operations/features should be required.

5.2. Dependencies

This implementation is constrained entirely to the LOD layer on the server and the lfs/llapi code on the client. The current design introduces minimal dependencies beyond those layers.

5.3. Security

This feature should have no security implications. All of its operations are contained within "setstripe" and normal file i/o operations, which already have existing security models which are not affected by this.

6. State

7. Use Cases

There are two primary use cases.

1. A system with multiple tiers of OSTs. This is, in essence, a hardware requirement to have different OST types and the desire to "spill over" from one set to another. The classic example is a tier of small, fast OSTs on flash, and then a second tier of larger, slower OSTs on spinning disk. Files would be created on flash and data would be written there either until the file reached an upper size limit or the OST(s) ran low on space. Then the layout would automatically "spill over" to the second tier of larger, slower OSTs on spinning disk. This should help users avoid ENOSPC errors when a particular OST or set OSTs fills up and there is still space free in another OST pool.

This gives more flexibility than a simple progressive file layout, such as 0-100 GB on flash, then 100 GB to infinity on disk. If flash runs out of space before 100GB, then writes will fail. A SEPFL layout might be 0-10GB, extending 10 GB at a time, upper limit of 200 GB, then to infinity on disk. If the flash tier gets low on space, the layout will automatically "spill over" to the disk tier.

2. When there is only one set of OSTs, the feature can still be used to help avoid ENOSPC. In this case, a self-extending layout can be created that only uses one tier. Something like: 0-100 GB, extending 100 GB, upper limit infinity. When a write occurs after 100 GiB, the layout will be extended on the current OSTs, if there is enough space. If there is not enough space, this will use the "self-spillover"/repeated component behavior of SEPFL. Another component is created using the striping information from the first component as a template, but with new striping requested from the stripe allocator. If there is sufficient free space on this component, the layout is extended on this component. This means that if one OST starts running out of space, the layout can adjust to use different OSTs, even in absence of explicit tiers.

As above, this would be a layout template that would be applied to files by default. This should reduce the impact of running low on space on a particular OST.

7.1 Scenarios

7.2 Failures

This feature does not introduce any obvious new failure-related issues - It uses the existing infrastructure for handling failure in the client-server connection (replay) and on the server (failover, fsck).

8. Analysis

8.1 Scalability

This feature is expected to have two scalability related concerns.

1. A larger number of layout components increases layout size, taking up more space on the MDT and requiring more time to send to the client. This is a problem common to other layout enhancements and to PFL itself. As the number of layout components rises, these requirements increase with it. There is no specific solution to this, but it is mitigated by not creating overly complex layouts. This problem is not expected to be serious as this feature will only occasionally create **new** layout components and otherwise has a marginal impact on the number of layout components vs a traditional PFL layout.
2. When client i/o reaches an extension space component, we must contact the MDT for additional layout in order to continue our I/O. This creates a delay on the client and work on the MDT. If clients must contact the MDT too often, the burden could become significant. The solution is simply to make the extension size (the amount of new layout granted) relatively large. There is a tradeoff - Larger extension sizes mean we must contact the metadata server less, but the granularity of space allocation is increased. This makes it more likely we will run out of space.

However, a little bit of simple math suggests this should be OK, even with relatively small extension sizes.

A very large file system will have between 1 TB/s and 10 TB/s of bandwidth (deliberately using very round numbers).

Consider an extension size of 100 GB at a time. If all files have this, that means when writing at at 1 TB/s we would do an average of 10 extensions (across all files) per second and 100 extensions per second when writing at 10 TB/s. That should have a totally negligible impact on the MDS. So perhaps 10 GB at a time is realistic - That would mean 100/s and 1000/s, respectively, when a 1 TB/s or 10 TB/s file system is at full speed. This assumes the entire file system is using this kind of file, but even then, 1000 is not a particularly large number of ops/second, and large file systems will probably have multiple MDSEs as well, so the load will be divided among them. These #s suggest this should not be a problem for reasonable extension sizes.

Both of these issues should be addressable by appropriate configuration.

8.2 Limitations

This section is intended to describe limitations to the expected functionality of the feature versus the design. This is NOT intended to describe intrinsic limitations of the design, rather it is intended to describe areas where the realized functionality falls short of the design due to outside factors.

There are two principal limitations that apply to this currently, both are limitations for progressive file layout s and are inherited here.

1. Appending to PFL files instantiates the entire layout. (LU-9341 at Intel) Currently, when a file is written to with O_APPEND, the entire layout must be instantiated. This is because knowing the current file size (required for append) requires locking all data components (so the file cannot be written to at a higher offset), and it is difficult to ensure no new data components will be created in a PFL file with uninstantiated components. The simple solution is to instantiate the entire layout.

That creates a write intent layout request from 0 to EOF. The current impact on SEPFL files is to extend all normal components to their upper limit and (as a result) remove all virtual/extension space components. There is a proposed fix in LU-9341, but no code to implement it yet.

2. Group locks have a similar issue, but the solution is distinct. Since a group lock is supposed to cover all of a file and be exclusive, it must lock all data components. As with appending, this requires fully instantiating the file. This has the same impact on SEPFL files as appending - It extends all components to their upper limit and virtual/extension space components. This is tracked in LU-9479. There is a proposal for a fix (essentially, group locks as layout locks), but it's just a rough idea and there is no code to implement it.

8.3 Testing

The implementation will include a number of Lustre sanity tests, exploring the basic functionality. These should cover basic functional testing, but there is also a requirement for system test to do scale testing, etc.

Some ideas for focus areas in system test include:

Many clients doing varied I/O to a file with a self-extending PFL layout (shared file writing (IOR), the write_disjoint test from the Lustre sanity suite, etc).

Testing with multiple pools of OSTs, filling one pool up and verifying spill over to the second works as expected

Testing of layout templates/default layouts, verifying inheritance and correct behavior (There will be at least one sanity test for this as well)

Testing of spillover behavior with multiple files & pools (Example: Create a SEPFL template/default layout to spill over from one pool to another, then write to multiple files until the first pool is low on space, and verify they spill over to the second pool successfully)

Generalized test runs with a self-extending PFL layout template set as the default layout. IE make all files without explicit striping in to SEPFL files, and run various tests.

Failover/failback testing while writing to self-extending PFL files. Verify no unexpected behavior.

8.4 Rationale

There are two main design elements with obvious alternatives.

1. The use of "virtual" components for extension space. We could instead have changed the behavior of space which had no layout defined at all. This would simplify things a bit conceptually for users - Virtual components are a bit odd. They really want to describe the behavior of the real component, and thinking about a virtual component is an extra burden which could be removed.

However, the virtual and uninitialized component has a number of advantages. It allows us to retain the existing behavior with uninitialized layouts, where a layout intent is sent to the server, the layout is refreshed, and i/o is restarted. This means clients that do not know about self-extending layouts can nevertheless read and write to them correctly. The virtual extension space component also gives us a place to store the extension space properties, such as its extent and the extension size. This allows us to avoid adding fields to the standard layout component, which would be unused for most components and waste storage space. This also lets us keep the on disk format the same, which eases version compatibility issues.

2. When extending the layout, we could either increase the dimensions of an existing component, or create a new layout component to cover the new space. The second option (which was the original design proposed in conversations at LAD) has a few advantages. In particular, it would be possible to address each component separately, for example with HSM release/restore. (This would require other changes, but it would be possible.) This is appealing, but it has a basic flaw that seems to make it impractical. Every time we add a layout component, it consumes space on the MDT and increases the layout size, which is eventually a performance problem. This means we want to keep the extension size relatively high, so we do not have to add too many components. But this is exactly the opposite of what we want if our goal is to use the components for partial release/restore. For that, we would like to be able to address relatively small pieces of the file, which means many components, which is highly impractical. Because we cannot satisfy the HSM case anyway, we chose the approach of increasing the dimensions of an existing component, at least where that's possible.

In the cases where extending the existing component would result in the client running out of space, we **do** attempt to create a new component (with new stripes) to allow the client to continue writing. But this is intended to be a rare situation, and should not create many components.

9. Deployment

Deployment will consist first of installing the required Lustre versions on client and server, and then using the feature. Actual use of the feature will require users/administrators to create layouts which use it. The intention is that default layouts making use of this feature will be created. The details of suggested layouts, etc, will be determined and documentation will be created to associate with the feature.

9.1 Compatibility

The current implementation **should** be mostly compatible with any client which understands progressive file layouts. There are currently **no** client implementation changes planned in Lustre itself to support this functionality. That may change as the implementation progresses, but that is the current state.

It is only "mostly" compatible because a client without this feature will not be able to correctly display self extending layouts. It will not recognize the extension flag or the special length descriptions. This does not impair correct operation because only the server uses these directly, but it could cause confusion.

Clients without this support would also be unable to create SEPFL layouts, since their lfs will not be able to express such a layout.

It may be best to prevent clients that do not understand SEPFL layouts from opening them, as is done with FLR files, but it may not be strictly necessary.

Server side compatibility is more of a problem. If we downgrade from a server with this support to one without it, the server will not handle the layout correctly. So we should make sure to use whatever compatibility mechanism exists for new types of layouts to ensure the server recognizes this and avoids trying to use them.

9.2 Installation

The component will be part of a new Lustre software package, which would be distributed in a NEO update. There are no special installation requirements.

10. References

FLR HLD:

http://wiki.lustre.org/File_Level_Replication_High_Level_Design

Layout Enhancement design:

http://wiki.lustre.org/Layout_Enhancement_High_Level_Design

PFL Design:

http://wiki.lustre.org/Progressive_File_Layouts

LU-10070: PFL self-extending file layout

<https://jira.hpdd.intel.com/browse/LU-10070>

LUS-2528: Spillover Space

LUS-2528 - Spillover space

IN PROGRESS