## dma-buf subsystem(一)\_Jun's blog

* Linux kernel 5.0 Document

# **1 Overview**

* Two definitions of dma-buf subsystem as the following comments described.
* (1) The **dma-buf subsystem** provides the framework for sharing buffers for hardware (DMA) access across multiple device drivers and subsystems, and for **synchronizing asynchronous hardware access**. This is used, for example, by **drm “prime”** multi-GPU support.
* (2) The **dma-buf framework** provides a generic method for sharing buffers between multiple devices. Device drivers that support dma-buf can export a DMA buffer to userspace as a file descriptor (known as the **exporter** role), import a DMA buffer from userspace using a file descriptor previously exported for a different or the same device (known as the **importer** role), or both.
* The **three main components** of dma-buf subsystem are:
  + (1) **dma-buf**, representing a **sg\_table** and exposed to userspace as a **file descriptor** to allow passing between devices.
  + (2) **fence**, which provides a mechanism to signal when one device as finished access.
  + (3) **reservation**, which manages the shared or exclusive fence(s) associated with the buffer.

# **2 Use the dma-buf**

## 2.1 exporter and importer

* Any device driver which wishes to be a part of DMA buffer sharing, can do so as either the ‘exporter’ of buffers, or the ‘importer’ of buffers. Say a driver A wants to use buffers created by driver B, then we call B as the exporter, and A as buffer-user/importer.
* **The exporter**
  + implements and manages operations in struct dma\_buf\_ops for the buffer,
  + allows other users to share the buffer by using dma\_buf sharing APIs,
  + manages the details of buffer allocation, wrapped in a struct dma\_buf,
  + decides about the actual backing storage where this allocation happens, and takes care of any migration of scatterlist - for all (shared) users of this buffer.
* **The buffer-user/importer**
  + is one of (many) sharing users of the buffer.
  + doesn’t need to worry about how the buffer is allocated, or where.
  + and needs a mechanism to get access to the scatterlist that makes up this buffer in memory, mapped into its own address space, so it can access the same area of memory. This interface is provided by struct dma\_buf\_attachment.
* Any exporters or users of the dma-buf buffer sharing framework must have a ‘select DMA\_SHARED\_BUFFER’ in their respective Kconfigs.

## 2.2 DMA buffer file descriptor

* Mostly a DMA buffer file descriptor is simply an opaque object for userspace, and hence the generic interface exposed is very minimal.
* userspace must have a way to request O\_CLOEXEC flag be set when the dma-buf fd is created.

## 2.3 Basic Operation and Device DMA Access

* The exporter defines his exporter instance using DEFINE\_DMA\_BUF\_EXPORT\_INFO()and calls dma\_buf\_export() to wrap a private buffer object into a dma\_buf. It then exports that dma\_buf to userspace as a file descriptor by calling dma\_buf\_fd().`
* Userspace passes this file-descriptors to all drivers it wants this buffer to share with: First the filedescriptor is converted to a dma\_buf using dma\_buf\_get(). Then the buffer is attached to the device using \*\*dma\_buf\_attach()\*\*.
* Once the buffer is attached to all devices userspace , then it can initiate DMA access to the shared buffer. In the kernel this is done by calling dma\_buf\_map\_attachment() and dma\_buf\_unmap\_attachment().

## 2.4 CPU Access to DMA Buffer Objects

### 2.4.1 fallback opeartions

* Fallback operations in the kernel, for example when a device is connected over USB and the kernel needs to shuffle the data around first before sending it away. Cache coherency is handled by braketing any transactions with calls to dma\_buf\_begin\_cpu\_access() and dma\_buf\_end\_cpu\_access() access.

"Strain operations in the kernel, such as devices connected via USB, the kernel needs to de-randomize the data before sending, then cache coherence is required for wrapping transfers, it (cache coherence) is through dma\_buf\_begin\_cpu\_access() and dma\_buf\_end\_cpu\_access() realized.

dma\_buf\_begin\_cpu\_access()

braketing any transactions

dma\_buf\_end\_cpu\_access()

2.4.2 vmap

* Since for most kernel internal dma-buf accesses need the entire buffer, a vmap interface is introduced.

// dma\_buf\_vmap - Create virtual mapping for the buffer object into kernel address space.

// This call may fail due to lack of virtual mapping address space

void \*dma\_buf\_vmap(struct dma\_buf \*dmabuf)

void dma\_buf\_vunmap(struct dma\_buf \*dmabuf, void \*vaddr)

# LWN Translation: DMA-BUF cache handling: Off the DMA API map (part 1)

Disclaimer: This article is not original, just a translation!  
Original text: <https://lwn.net/Articles/822521/>  
Author: John Stultz (Linaro member, kernel timekeeping maintainer)  
Note: This article requires background knowledge of [DMA](https://so.csdn.net/so/search?q=DMA&spm=1001.2101.3001.7020) -BUF. If you don’t know DMA-BUF yet, it is recommended to read it first The translator's own ["dma-buf from shallow to deep"](https://blog.csdn.net/hexiaolong2009/category_10838100.html)[series](https://blog.csdn.net/hexiaolong2009/article/details/102596772) Chapter 3 "map attachment" [and](https://blog.csdn.net/hexiaolong2009/article/details/102596825) Chapter 6 "begin / end cpu\_access" .

In the [previous](https://blog.csdn.net/hexiaolong2009/article/details/106745686) article, I introduced some background knowledge about ION, DMA-BUF Heap, DMA [API](https://so.csdn.net/so/search?q=API&spm=1001.2101.3001.7020) , and the basic concept of CPU Cache "ownership", and finally described how DMA-BUF is from the perspective of traditional DMA API Deals with cache synchronization issues. The article concludes by discussing why traditional DMA APIs perform so poorly on modern mobile platforms. This article will discuss with you how to make the DMA-BUF exporter avoid unnecessary cache operations, and give some general suggestions on how to improve these methods.

From the point of view of the DMA API: by calling *dma\_buf\_map\_attachment()* , the ownership of the DMA-BUF is transferred to the DMA device, and by calling *dma\_buf\_unmap\_attachment()* , the ownership is returned to the CPU, and each time these two functions are called, the cache correlation is executed operation. Although such a sequential operation can ensure the correctness of CPU Cache processing, for buffer pipeline operations involving multiple DMA devices, the CPU does not actually participate in accessing these buffers at all, and each cache map and unmap operations increase It can cause significant performance issues.

**Who owns the buffer?**

To avoid these redundant cache operations, the DMA-BUF interface allows some rules of the DMA API to be reversed. It should be noted that the DMA API assumes that the CPU is the natural owner of all memory, and this reverse rule only needs to be considered during a DMA transfer (the ownership of the buffer has been explicitly transferred to the DMA device). [*The DMA-BUF interface requires the CPU to call dma\_buf\_begin\_cpu\_access()*](https://elixir.bootlin.com/linux/v5.7.1/source/drivers/dma-buf/dma-buf.c#L1064) before accessing the DMA-BUF, and call *[dma\_buf\_end\_cpu\_access](https://elixir.bootlin.com/linux/v5.7.1/source/drivers/dma-buf/dma-buf.c" \l "L1100)* () after the access ends . If the CPU wants to access the buffer from user space, it can use the DMA\_BUF\_IOCTL\_SYNCioctl() command to initiate a call to begin/end cpu\_access.

Special interface:

* *dma\_buf\_begin\_cpu\_access()*  
  Through this interface, the exporter driver can ensure that the current buffer is only allowed to be accessed by the CPU. In this process, allocate or swap-in and pin (fixed) backend storage may be required. In addition, the exporter driver also needs to ensure that the direction of the CPU access is consistent with the direction it requests.
* *dma\_buf\_end\_cpu\_access()*  
  This interface is called when the importer completes the CPU Access. The exporter can implement the cache flush operation in this interface and unpin the memory resources pinned in *dma\_buf\_begin\_cpu\_access() .*

When the above interfaces are used, we can think that the DMA-BUF memory belongs to the DMA device by default, not the CPU. Therefore, it is necessary to complete the synchronization operation of the CPU Cache in these interfaces to ensure that the data obtained by the CPU is consistent with that in the DMA-BUF. At the same time, this method can also avoid the expensive cache synchronization operations caused by only passing, mapping and accessing DMA-BUFs between multiple devices.

However, this inconsistent calling rule with the DMA API may cause some confusion, and not all DMA-BUF exporter drivers use the same implementation strategy. Some exporter drivers intend to still follow the DMA API calling rules, flush and invalidate the CPU cache every time the map and unmap operations are performed; other exporter drivers may only be performed in their begin and end callback interfaces. Cache synchronization operation, and some exporter drivers may implement both solutions.

Although DMA BUF is designed to share memory between user space and multiple DMA devices, the exporter that first exports DMA-BUF is often a special driver, which is customized by the manufacturer and strongly related to the driver. The buffer allocation code. For example, a GPU driver that allocates a buffer, then performs rendering operations on it, and returns a handle to user space. The user-space application can then send this buffer, along with other buffers, back to the GPU to composite the web browser window with other windows on the desktop. DMA-BUF provides a more general handle type, so even if the buffer is not used for multi-device sharing, its handle can still be used.

However, it should be known that the cache synchronization operation needs to be considered only when the buffer is shared between the CPU and the DMA device, so the DMA-BUF exporter can do some cache optimization for the case where the buffer is only shared between multiple devices. For example, some DMA-BUF exporter drivers first save the scatter-gather table when performing the DMA mapping operation for the first time, and continue to use these tables as long as subsequent *dma\_buf\_map\_attachment()* calls are executed in the same DMA direction . In this way, we can avoid the expensive cache operation every time we call *dma\_buf\_map\_attachment()* and *dma\_buf\_unmap\_attachment() , and finally release the previous DMA mapping resources in dma\_buf\_detach()* . These optimizations work because the exporter is bound to the DMA device, so the buffer is not actually shared, or the DMA devices that share the buffer are all cache consistent, so there is no need to maintain the cache. operation.

Translator's Note: The above paragraph is really difficult to translate, especially the boldface part below, it is really difficult to understand, so I will post the original text, hoping that someone can correct the mistake, so as not to mislead the children.  
  
  
***But, knowing that the buffer was shared between just the CPU and the device, the DMA-BUF exporter could optimize some of the cache operations.*** For instance, some DMA-BUF exporters cache the scatter/gather table resulting from the first DMA mapping operation and, as long as the dma\_buf\_map\_attachment() calls are done in the same direction, reuse that table. In this way, they can avoid expensive cache operations on each dma\_buf\_map\_attachment() and dma\_buf\_unmap\_attachment() call, finally releasing the mapping in dma\_buf\_detach() . ***These optimizations work because the exporters are tied to the device, so the buffers aren't really being shared,***or the devices the buffers are shared with are cache coherent, so the cache maintenance is unnecessary.

1. The first sentence clearly states that the buffer is shared between the CPU and the device. In this case, the cache operation must be performed every time. How to optimize it?
2. Why does the second sentence say that the exporter is bound to the device? Why is the buffer not really shared? Sharing buffers between multiple DMA devices is not called sharing?

Although this method is effective, it results in that in the upstream version, more than a dozen DMA-BUF exporter drivers have their own different cache processing methods and calling rules. Therefore, when we started to study how to implement a general DMA-BUF exporter framework to support multi-device pipeline from a certain performance point of view, we could not find a clear implementation solution.

**Handling buffer ownership issues with multiple mappings**

While the DMA API provides good documentation on how to use the map and unmap calls (to specify buffer ownership), achieving good performance on mobile platforms often requires multiple DMA devices and CPUs to establish valid buffers at the same time. mapping, which makes the concept of buffer ownership more subtle. For example, in a graphics system, the GPU and Display are usually mapped to the same buffer at the same time. For this reason, the system must establish a framebuffer sharing mapping relationship between multiple devices before the frame is drawn. In this way, the GPU can directly write data to the buffer, and then send a signal to the display driver after the writing is completed, and then the display driver can display the buffer immediately.

For this specific application scenario, DMA-BUF adds [dma-fence based on the](https://www.kernel.org/doc/html/v5.6/driver-api/dma-buf.html" \l "dma-fences)[explicit fence](https://lwn.net/Articles/702339/) architecture , which provides a mechanism for the driver (or user space) to wait for buffer fence. Eventually, another driver will signal the fence, thereby starting the switch of buffer ownership. However, to support this parallel mapping relationship requires careful handling of cache synchronization issues, which are usually implemented by the driver calling the DMA API synchronization interface. When a developer develops with a vendor-specific kernel on an integrated device, he may know which driver a buffer is coming from and to whom, so he can add the most appropriate and correct cache processing code. But once it's beyond his control, things get pretty complicated.

So we see here that there are two different ways of handling *ownership* tracking. **Implicit** handling means that the ownership of the DMA-BUF is switched when dma map or unmap, and **explicit** handling means that the buffer has been mapped to two or more devices, it Ownership is effectively switched through DMA-BUF fence.

The DMA-BUF exporter usually handles cache related operations while passing buffer ownership. They can do this in the implicit context of calls to *dma\_buf\_map\_attachment()* and *dma\_buf\_unmap\_attachment() , or they can do so in calls to dma\_buf\_begin\_cpu\_access()* and *dma\_buf\_end\_cpu\_access()* . However, in the case of explicit handling, the DMA-BUF exporter does not have a callback interface for DMA-BUF fence signals, so the exporter cannot perform any cache management operations for ownership switching, which creates a dilemma. In this case, the responsibility of buffer cache management is allocated to the DMA-BUF exporter and the driver using the buffer. To do this correctly, each driver must understand its position in the buffer pipeline and thus the cache coherency of its downstream devices.

Even more troublesome, even if the DMA-BUF exporter does have a callback interface for the dma-fence signal, it has no way of knowing which ownership tracking method is currently in use. Assuming the explicit processing mode defaults to CPU ownership, do we perform cache operations in the map and unmap functions? Or the implicit handling mode defaults to device ownership and we do cache operations in *dma\_buf\_begin\_cpu\_access()* and *dma\_buf\_end\_cpu\_access() ?*Or do we avoid the extra cache overhead when the driver switches ownership by executing an explicit fence signal? These choices may leave us with an implementation that is either too slow to use, or may be incompatible with some drivers, which completely defeats the original purpose of DMA BUF as a general purpose swap mechanism.

So to a developer trying to write a DMA-BUF exporter driver, this all starts to feel like a [level](https://ozlabs.org/~rusty/ols-2003-keynote/img56.html) 10 ("read the documentation, you'll get it wrong") or level 11 ("follow the routine, you'll get it wrong"), especially if you care about performance. This presents a huge obstacle to the goal of sharing a common DMA-BUF Heap among vendors.

**possible solution**

I think we can improve this situation, and I have some ideas that I can share with you. Since the DMA-BUF interface has deviated from the DMA API, I think we should establish some clear specifications for the use of DMA-BUF, and form a good development document, so that DMA-BUF exporter authors and DMA-BUF users can understand The model has a unified understanding. We should focus on the following directions:

* Creates a formal ownership of the DMA-BUF object outside the implicit map/unmap functions of the DMA API.
* Provide a set of calling mechanisms to track ownership, these interfaces can be added to the *dma buf\_ops* structure, so that the exporter driver can know the state changes of these ownerships.
* Implicit handling mode is deprecated and drivers should be asked to use the new mechanism above to mark ownership switches in explicit handling mode.
* Add some state tracking interfaces to DMA-BUFs, so that we can know their cache state, and only perform corresponding cache operations when ownership switches, so those state tracking interfaces become particularly important.

Most of the above can be achieved by documenting and enhancing the current DMA-BUF exporter invocation mechanism. *The dma\_buf\_begin\_cpu\_access()* and *dma\_buf\_end\_cpu\_access()* calls are sufficient to handle device-to-CPU and CPU-to-device transitions. But we need to clearly define the correct usage specification of these functions, and should always be implemented by the DMA-BUF exporter driver, thus normalizing the notion that buffers are device-owned by default. This way you can safely implement pre-flushed buffers and skip unnecessary cache operations.

However, this method has a disadvantage. For the case that the CPU needs to access the buffer multiple times (the device is not involved in the middle), each call will have an unnecessary cache flush operation. In addition, there is a problem that for a hybrid system with both CPU-coherent and non-coherent devices participating, we may need to do CPU-cache synchronization when switching ownership between these devices. In both cases, it might be helpful to use the device-usage function call and state tracking interface, so that you can decide whether to switch ownership (rather than just use it).

This concept of *ownership* also needs to take into account future partial cache flush operations to allow the CPU and DMA devices to access the same buffer at the same time. In this way, buffer ownership (and related cache operations) will be managed at the granularity of individual cache lines, rather than at the level of the entire buffer, which looks more like advisory range locks on file operations .

It is undeniable that DMA-BUF Heap (and ION before it) in some cases, the user space will know more about the purpose of the buffer than the kernel space. Therefore, it is most appropriate to let user space choose the buffer allocation type for a pipeline. The DMA-BUF design philosophy provides us with very practical flexibility, which allows to leave buffer rules and policies to the exporter driver, so I don't want to eliminate that flexibility. But I do think that as vendor vendors start their ION migration work, it is more important to have a clear and established specification, so that everyone will not fall into the pit, so as to avoid a batch of unnecessary , incompatible heaps and consumers. I hope this article can be a source of inspiration and arouse further discussion.

**thanks**

Many thanks to Rob Clark, Robert Foss, Sumit Semwal, Azam Sadiq Pasha kapatral Syed, Daniel Vetter and Linus Walleij for their early reviews and feedback on these two articles!

Previous: ["LWN translation: DMA-BUF cache handling: Off the DMA API map (part 1)"](https://blog.csdn.net/hexiaolong2009/article/details/106745686)

**DMA-BUF Article Summary:**["My DMA-BUF Column"](https://blog.csdn.net/hexiaolong2009/category_10838100.html)

# LWN translation: DMA-BUF cache handling: Off the DMA API map (part 2)

Disclaimer: This article is not original, just a translation!  
Original text: <https://lwn.net/Articles/822521/>  
Author: John Stultz (Linaro member, kernel timekeeping maintainer)  
Note: This article requires background knowledge of [DMA](https://so.csdn.net/so/search?q=DMA&spm=1001.2101.3001.7020) -BUF. If you don’t know DMA-BUF yet, it is recommended to read it first The translator's own ["dma-buf from shallow to deep"](https://blog.csdn.net/hexiaolong2009/category_10838100.html)[series](https://blog.csdn.net/hexiaolong2009/article/details/102596772) Chapter 3 "map attachment" [and](https://blog.csdn.net/hexiaolong2009/article/details/102596825) Chapter 6 "begin / end cpu\_access" .

In the [previous](https://blog.csdn.net/hexiaolong2009/article/details/106745686) article, I introduced some background knowledge about ION, DMA-BUF Heap, DMA [API](https://so.csdn.net/so/search?q=API&spm=1001.2101.3001.7020) , and the basic concept of CPU Cache "ownership", and finally described how DMA-BUF is from the perspective of traditional DMA API Deals with cache synchronization issues. The article concludes by discussing why traditional DMA APIs perform so poorly on modern mobile platforms. This article will discuss with you how to make the DMA-BUF exporter avoid unnecessary cache operations, and give some general suggestions on how to improve these methods.

From the point of view of the DMA API: by calling *dma\_buf\_map\_attachment()* , the ownership of the DMA-BUF is transferred to the DMA device, and by calling *dma\_buf\_unmap\_attachment()* , the ownership is returned to the CPU, and each time these two functions are called, the cache correlation is executed operation. Although such a sequential operation can ensure the correctness of CPU Cache processing, for buffer pipeline operations involving multiple DMA devices, the CPU does not actually participate in accessing these buffers at all, and each cache map and unmap operations increase It can cause significant performance issues.

**Who owns the buffer?**

To avoid these redundant cache operations, the DMA-BUF interface allows some rules of the DMA API to be reversed. It should be noted that the DMA API assumes that the CPU is the natural owner of all memory, and this reverse rule only needs to be considered during a DMA transfer (the ownership of the buffer has been explicitly transferred to the DMA device). [*The DMA-BUF interface requires the CPU to call dma\_buf\_begin\_cpu\_access()*](https://elixir.bootlin.com/linux/v5.7.1/source/drivers/dma-buf/dma-buf.c#L1064) before accessing the DMA-BUF, and call *[dma\_buf\_end\_cpu\_access](https://elixir.bootlin.com/linux/v5.7.1/source/drivers/dma-buf/dma-buf.c" \l "L1100)* () after the access ends . If the CPU wants to access the buffer from user space, it can use the DMA\_BUF\_IOCTL\_SYNCioctl() command to initiate a call to begin/end cpu\_access.

Special interface:

* *dma\_buf\_begin\_cpu\_access()*  
  Through this interface, the exporter driver can ensure that the current buffer is only allowed to be accessed by the CPU. In this process, allocate or swap-in and pin (fixed) backend storage may be required. In addition, the exporter driver also needs to ensure that the direction of the CPU access is consistent with the direction it requests.
* *dma\_buf\_end\_cpu\_access()*  
  This interface is called when the importer completes the CPU Access. The exporter can implement the cache flush operation in this interface and unpin the memory resources pinned in *dma\_buf\_begin\_cpu\_access() .*

When the above interfaces are used, we can think that the DMA-BUF memory belongs to the DMA device by default, not the CPU. Therefore, it is necessary to complete the synchronization operation of the CPU Cache in these interfaces to ensure that the data obtained by the CPU is consistent with that in the DMA-BUF. At the same time, this method can also avoid the expensive cache synchronization operations caused by only passing, mapping and accessing DMA-BUFs between multiple devices.

However, this inconsistent calling rule with the DMA API may cause some confusion, and not all DMA-BUF exporter drivers use the same implementation strategy. Some exporter drivers intend to still follow the DMA API calling rules, flush and invalidate the CPU cache every time the map and unmap operations are performed; other exporter drivers may only be performed in their begin and end callback interfaces. Cache synchronization operation, and some exporter drivers may implement both solutions.

Although DMA BUF is designed to share memory between user space and multiple DMA devices, the exporter that first exports DMA-BUF is often a special driver, which is customized by the manufacturer and strongly related to the driver. The buffer allocation code. For example, a GPU driver that allocates a buffer, then performs rendering operations on it, and returns a handle to user space. The user-space application can then send this buffer, along with other buffers, back to the GPU to composite the web browser window with other windows on the desktop. DMA-BUF provides a more general handle type, so even if the buffer is not used for multi-device sharing, its handle can still be used.

However, it should be known that the cache synchronization operation needs to be considered only when the buffer is shared between the CPU and the DMA device, so the DMA-BUF exporter can do some cache optimization for the case where the buffer is only shared between multiple devices. For example, some DMA-BUF exporter drivers first save the scatter-gather table when performing the DMA mapping operation for the first time, and continue to use these tables as long as subsequent *dma\_buf\_map\_attachment()* calls are executed in the same DMA direction . In this way, we can avoid the expensive cache operation every time we call *dma\_buf\_map\_attachment()* and *dma\_buf\_unmap\_attachment() , and finally release the previous DMA mapping resources in dma\_buf\_detach()* . These optimizations work because the exporter is bound to the DMA device, so the buffer is not actually shared, or the DMA devices that share the buffer are all cache consistent, so there is no need to maintain the cache. operation.

Translator's Note: The above paragraph is really difficult to translate, especially the boldface part below, it is really difficult to understand, so I will post the original text, hoping that someone can correct the mistake, so as not to mislead the children.  
  
  
***But, knowing that the buffer was shared between just the CPU and the device, the DMA-BUF exporter could optimize some of the cache operations.*** For instance, some DMA-BUF exporters cache the scatter/gather table resulting from the first DMA mapping operation and, as long as the dma\_buf\_map\_attachment() calls are done in the same direction, reuse that table. In this way, they can avoid expensive cache operations on each dma\_buf\_map\_attachment() and dma\_buf\_unmap\_attachment() call, finally releasing the mapping in dma\_buf\_detach() . ***These optimizations work because the exporters are tied to the device, so the buffers aren't really being shared,***or the devices the buffers are shared with are cache coherent, so the cache maintenance is unnecessary.

1. The first sentence clearly states that the buffer is shared between the CPU and the device. In this case, the cache operation must be performed every time. How to optimize it?
2. Why does the second sentence say that the exporter is bound to the device? Why is the buffer not really shared? Sharing buffers between multiple DMA devices is not called sharing?

Although this method is effective, it results in that in the upstream version, more than a dozen DMA-BUF exporter drivers have their own different cache processing methods and calling rules. Therefore, when we started to study how to implement a general DMA-BUF exporter framework to support multi-device pipeline from a certain performance point of view, we could not find a clear implementation solution.

**Handling buffer ownership issues with multiple mappings**

While the DMA API provides good documentation on how to use the map and unmap calls (to specify buffer ownership), achieving good performance on mobile platforms often requires multiple DMA devices and CPUs to establish valid buffers at the same time. mapping, which makes the concept of buffer ownership more subtle. For example, in a graphics system, the GPU and Display are usually mapped to the same buffer at the same time. For this reason, the system must establish a framebuffer sharing mapping relationship between multiple devices before the frame is drawn. In this way, the GPU can directly write data to the buffer, and then send a signal to the display driver after the writing is completed, and then the display driver can display the buffer immediately.

For this specific application scenario, DMA-BUF adds [dma-fence based on the](https://www.kernel.org/doc/html/v5.6/driver-api/dma-buf.html" \l "dma-fences)[explicit fence](https://lwn.net/Articles/702339/) architecture , which provides a mechanism for the driver (or user space) to wait for buffer fence. Eventually, another driver will signal the fence, thereby starting the switch of buffer ownership. However, to support this parallel mapping relationship requires careful handling of cache synchronization issues, which are usually implemented by the driver calling the DMA API synchronization interface. When a developer develops with a vendor-specific kernel on an integrated device, he may know which driver a buffer is coming from and to whom, so he can add the most appropriate and correct cache processing code. But once it's beyond his control, things get pretty complicated.

So we see here that there are two different ways of handling *ownership* tracking. **Implicit** handling means that the ownership of the DMA-BUF is switched when dma map or unmap, and **explicit** handling means that the buffer has been mapped to two or more devices, it Ownership is effectively switched through DMA-BUF fence.

The DMA-BUF exporter usually handles cache related operations while passing buffer ownership. They can do this in the implicit context of calls to *dma\_buf\_map\_attachment()* and *dma\_buf\_unmap\_attachment() , or they can do so in calls to dma\_buf\_begin\_cpu\_access()* and *dma\_buf\_end\_cpu\_access()* . However, in the case of explicit handling, the DMA-BUF exporter does not have a callback interface for DMA-BUF fence signals, so the exporter cannot perform any cache management operations for ownership switching, which creates a dilemma. In this case, the responsibility of buffer cache management is allocated to the DMA-BUF exporter and the driver using the buffer. To do this correctly, each driver must understand its position in the buffer pipeline and thus the cache coherency of its downstream devices.

Even more troublesome, even if the DMA-BUF exporter does have a callback interface for the dma-fence signal, it has no way of knowing which ownership tracking method is currently in use. Assuming the explicit processing mode defaults to CPU ownership, do we perform cache operations in the map and unmap functions? Or the implicit handling mode defaults to device ownership and we do cache operations in *dma\_buf\_begin\_cpu\_access()* and *dma\_buf\_end\_cpu\_access() ?*Or do we avoid the extra cache overhead when the driver switches ownership by executing an explicit fence signal? These choices may leave us with an implementation that is either too slow to use, or may be incompatible with some drivers, which completely defeats the original purpose of DMA BUF as a general purpose swap mechanism.

So to a developer trying to write a DMA-BUF exporter driver, this all starts to feel like a [level](https://ozlabs.org/~rusty/ols-2003-keynote/img56.html) 10 ("read the documentation, you'll get it wrong") or level 11 ("follow the routine, you'll get it wrong"), especially if you care about performance. This presents a huge obstacle to the goal of sharing a common DMA-BUF Heap among vendors.

**possible solution**

I think we can improve this situation, and I have some ideas that I can share with you. Since the DMA-BUF interface has deviated from the DMA API, I think we should establish some clear specifications for the use of DMA-BUF, and form a good development document, so that DMA-BUF exporter authors and DMA-BUF users can understand The model has a unified understanding. We should focus on the following directions:

* Creates a formal ownership of the DMA-BUF object outside the implicit map/unmap functions of the DMA API.
* Provide a set of calling mechanisms to track ownership, these interfaces can be added to the *dma buf\_ops* structure, so that the exporter driver can know the state changes of these ownerships.
* Implicit handling mode is deprecated and drivers should be asked to use the new mechanism above to mark ownership switches in explicit handling mode.
* Add some state tracking interfaces to DMA-BUFs, so that we can know their cache state, and only perform corresponding cache operations when ownership switches, so those state tracking interfaces become particularly important.

Most of the above can be achieved by documenting and enhancing the current DMA-BUF exporter invocation mechanism. *The dma\_buf\_begin\_cpu\_access()* and *dma\_buf\_end\_cpu\_access()* calls are sufficient to handle device-to-CPU and CPU-to-device transitions. But we need to clearly define the correct usage specification of these functions, and should always be implemented by the DMA-BUF exporter driver, thus normalizing the notion that buffers are device-owned by default. This way you can safely implement pre-flushed buffers and skip unnecessary cache operations.

However, this method has a disadvantage. For the case that the CPU needs to access the buffer multiple times (the device is not involved in the middle), each call will have an unnecessary cache flush operation. In addition, there is a problem that for a hybrid system with both CPU-coherent and non-coherent devices participating, we may need to do CPU-cache synchronization when switching ownership between these devices. In both cases, it might be helpful to use the device-usage function call and state tracking interface, so that you can decide whether to switch ownership (rather than just use it).

This concept of *ownership* also needs to take into account future partial cache flush operations to allow the CPU and DMA devices to access the same buffer at the same time. In this way, buffer ownership (and related cache operations) will be managed at the granularity of individual cache lines, rather than at the level of the entire buffer, which looks more like advisory range locks on file operations .

It is undeniable that DMA-BUF Heap (and ION before it) in some cases, the user space will know more about the purpose of the buffer than the kernel space. Therefore, it is most appropriate to let user space choose the buffer allocation type for a pipeline. The DMA-BUF design philosophy provides us with very practical flexibility, which allows to leave buffer rules and policies to the exporter driver, so I don't want to eliminate that flexibility. But I do think that as vendor vendors start their ION migration work, it is more important to have a clear and established specification, so that everyone will not fall into the pit, so as to avoid a batch of unnecessary , incompatible heaps and consumers. I hope this article can be a source of inspiration and arouse further discussion.

**thanks**

Many thanks to Rob Clark, Robert Foss, Sumit Semwal, Azam Sadiq Pasha kapatral Syed, Daniel Vetter and Linus Walleij for their early reviews and feedback on these two articles!

Previous: ["LWN translation: DMA-BUF cache handling: Off the DMA API map (part 1)"](https://blog.csdn.net/hexiaolong2009/article/details/106745686)

**DMA-BUF Article Summary:**["My DMA-BUF Column"](https://blog.csdn.net/hexiaolong2009/category_10838100.html)