Shared images and synchronization

AKA Mailbox replacement

piman@chromium.org

Last update: 2019-05-09
Status: accepted, implementation in progress
Self-link: tinyurl.com/chrome-shared-images

Background

GPU mailboxes

Mailboxes are a mechanism to share GL textures between different client-side GLES2/command-buffer contexts, including across multiple processes. Except for Android WebView, they are a pure command buffer-level abstraction, and as such have very well defined semantics that is trivially portable across platforms. Basically the underlying driver-level texture ("**`service_id`**") is mapped as a client-level texture ("**`client_id`**") in each of the participating clients, including the structures (**`gpu::TextureBase`** derived types) tracking structural properties (target, size, format, etc.) and state (e.g. texture filters). **`gpu::MailboxManager`** keeps track of a mapping from each **`gpu::Mailbox`** (an unguessable identifier) to the corresponding **`gpu::TextureBase`**. This relies on all service-side contexts to be in the same global share group so that they can all access the same texture. Synchronization between clients directly relies on command-level ordering, provided by OrderingBarriers and SyncTokens, where it is assumed, as a fundamental property, that GL commands sent to multiple driver-level contexts will be sequentially ordered with respect to each other. When this is not guaranteed at the platform level, we rely on virtualized contexts to provide that guarantee. This also relies on all commands executing on a single thread, because the tracking structures are fundamentally not thread-safe.

Android WebView needs to execute GL commands on 2 separate threads. On that particular platform, to keep mailbox semantics as close as possible to other platforms, we use a special implementation of **`gpu::MailboxManager`**, **`gpu::MailboxManagerSync`**, that attempts to synchronize texture image data and state between 2 different GL textures in 2 different driver-level contexts/share groups, as well as serializing accesses to those textures from the different threads. It does so by mapping one texture's base image level to an **`EGLImage`**, and binding that **`EGLImage`** to the other texture's base image level, as well as updating state (texture parameters) across. This tracking/updating is done at synchronization points (sync tokens) in each context. This code is fairly complex (leading to bugs), and has some limitations relative to "regular" mailboxes, namely that only the first texture level is handled (others are simply ignored), and only a limited number of texture types (2D ones only) are supported. It is also fairly costly, because, on each sync token operation, every texture shared this way needs to be checked for updates, which scales linearly with the number of shared (not updated) textures.

Overall this system is very flexible for the client-side, permitting sharing of arbitrary textures across clients, and a multitude of access patterns (e.g. producer/consumer, multiple concurrent reads, arbitrary interleaving of accesses), while guaranteeing well-defined semantics (although possibly unpredictable if synchronization is absent or incorrect). However, this flexibility comes

with strong constraints on the implementation side, which have performance costs (virtualized contexts, lack of threading capabilities, synchronization overhead in Android WebView), and have a major fundamental restriction, which is that it is entirely based on GL.

To be able to support the Vulkan API to back the majority of rendering in Chrome, especially on platforms that will not support GL natively (Fuchsia), and want to enable the use of multiple threads, we need to evolve the system into something that imposes fewer constraints on the implementation side, while maintaining enough flexibility that Chrome use case needs are met.

GL_CHROMIUM_image / GpuMemoryBuffer

A related way of sharing textures across clients, including the CPU, is via **`GpuMemoryBuffer`** objects, each of which abstracts a GPU-visible section of memory. That memory is created with size, format, and usage, and is represented in a platform-specific way by an IPC-compatible handle. This object can also be bound to a GL texture via the **`GL_CHROMIUM_image`** extension. Note that there are 2 ways of sharing these objects, one via duplicating/sending the **`GpuMemoryBuffer`** handle over IPC, and another one by using mailboxes to share the GL texture it is bound to.

`GpuMemoryBuffers` have multiple advantages over regular GL textures: they can reduce the number of copies between the CPU and the GPU, and they allow the memory to be shared between devices other than the GPU, such as video decoders, display controllers, or other platform-level APIs (e.g. system compositor). Depending on the usage pattern, they need explicit synchronization on the service side (e.g. to synchronize between those multiple devices) and possibly on the client-side (in particular for GPU vs CPU hazards). They also have constraints, such as immutability, single image (no mipmapping), a limited number of formats, platform-specific logic.

To the extent that it's possible, we want to keep full **`GpuMemoryBuffer`** support for Vulkan. This means being able to import them into the Vulkan API, and synchronizing access to the underlying memory.

Constraints

Vulkan / GL interoperability extensions

Vulkan and GL define a number of extensions, which, when used in combination, allow sharing of GPU memory (and corresponding resources) as well as synchronization primitives, between

both APIs, as well as other platform-specific APIs (e.g other 3D APIs, video decoding APIs, display/compositing APIs).

Those consist of (mostly) cross-platform base extensions that introduce the concepts and mechanisms, and various platform-specific layered extensions that expose the functionality using native handles/APIs. There are a few variations, mainly around whether the memory is allocated by one API (typically Vulkan), exported from that API, then imported into another API, or created using platform-specific mechanisms and then imported into all the relevant APIs.

The bottom line however is that it is never possible to import an existing GL texture into Vulkan.

Vulkan/GL external memory and semaphores

This is a generic API to share between Vulkan and GL using opaque native handles. This is expected to be available on Desktop (Windows, Linux), and although possible in theory on Android, the Android-specific API (based on **`AHardwareBuffer`**, see below) is expected to be more widely available (and tested. Well, maybe.).

In a nutshell, memory is created in Vulkan using traditional mechanisms, except with additional parameters that allow for sharing, then the memory is exported using an opaque native handle (file descriptor on Linux/Android, **`HANDLE`** on Windows). GL extensions allow importing the handles into GL, to create a memory object, that can be used to back textures. Similarly, synchronization between the APIs is done by creating a semaphore in Vulkan using traditional mechanisms (again with additional parameters), and exported using an opaque native handle, then imported into GL.

Cross-platform base:
VK_KHR_external_memory
VK_KHR_external_semaphore
GL_EXT_memory_object and GL_EXT_semaphore

Linux (incl. Android) specifics:
VK_KHR_external_memory_fd
VK_KHR_external_semaphore_fd
GL_EXT_memory_object_fd and GL_EXT_semaphore_fd
Using handle types
`VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT` and
`VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT`

Windows specifics:

VK_KHR_external_memory_win32
VK_KHR_external_semaphore_win32
GL_EXT_memory_object_win32 and GL_EXT_semaphore_win32
Using handle types **VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT** or **VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT** and **VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT** or **VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT**

Native buffers + Vulkan external objects + **EGLImage**/**EGLSync** (Android **AHardwareBuffer** + sync fd, Chrome OS DMA-BUF + sync fd):

This API assumes that the client creates or obtains a native buffer using platform-specific APIs (or possibly, Vulkan or EGL themselves), that is then imported into Vulkan, using the external memory API, and imported into GL via EGL, using **EGLImage** (like various **GLImageEGL** subclasses implement today). Synchronization is also done via platform-specific synchronization objects (kernel sync fd for Android and Chrome OS), which are exported out of an API-specific object (**VkSemaphore** or **EGLSync**) on the signaling side, and imported into another API-specific object on the waiting side. The native buffer and synchronization object can also be passed to other platform-specific APIs, e.g. video decoding or display. This path is, modulo potential format restrictions, compatible with **GpuMemoryBuffer**s.

Cross-platform base:
VK_KHR_external_memory
VK_KHR_external_semaphore
EGL_KHR_image_base
EGL_KHR_fence_sync
EGL_KHR_wait_sync
GL_OES_EGL_image_external
GL_OES_EGL_image

Android specifics:
VK_KHR_external_semaphore_fd
VK_ANDROID_external_memory_android_hardware_buffer
EGL_ANDROID_get_native_client_buffer
EGL_ANDROID_native_fence_sync
Using handle types
VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID and **VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT**

Chrome OS specifics:

VK_KHR_external_semaphore_fd
VK_EXT_external_memory_dma_buf (+ not-yet-specced modifier extension)
EGL_EXT_image_dma_buf_import
EGL_EXT_image_dma_buf_import_modifiers
EGL_ANDROID_native_fence_sync
Using handle types **VK_EXTERNAL_MEMORY_HANDLE_TYPE_DMA_BUF_BIT_EXT** and **VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT**

Native buffers + Vulkan external objects + EGL PBuffers (Windows D3D11 texture / DXGI shared handle + DXGI KeyedMutex):

This API is conceptually fairly similar to the previous version but uses a different mechanism to import into GL (via EGL **PBuffer**s rather than **EGLImage**s), and a different synchronization mechanism (DXGI KeyedMutex).

Cross-platform base:
VK_KHR_external_memory
Section “3.6 Rendering to Textures” in EGL

Windows specifics:
VK_KHR_external_memory_win32
VK_KHR_win32_keyed_mutex
EGL_ANGLE_d3d_texture_client_buffer
EGL_ANGLE_keyed_mutex

Using handle types
VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT or
VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT

Concurrent reads

More than one reader may access the image at the same time, e.g. canvas (or WebGL) can read a video (or other canvas or WebGL frontbuffer) that is being displayed.
Conceptual “read” operations may “write” on the state of the back-end image. E.g. layout and queue transitions in Vulkan, sampler parameters (filter/wrap modes) in GL.
May need to track state and/or take exclusive locks on the service side.

Threading

When possible, more than one thread may want to access a resource. Begin/End can be used to synchronize across them. TODO details.

Immutability

Unrelated clients modifying structure (size/format/etc.) involve extensive tracking service-side (e.g. tracking valid bounds for FBOs), and spec complexity (texture/FBO becoming “incomplete” with no visible change). Most backends can’t really support it anyway (e.g. Vulkan objects are immutable, **`glEGLImageTargetTexture2DOES`**-bound textures have side effects when mutated, etc.). Removing support for mutability simplifies the design, as long as clients can accommodate that constraint. It also significantly simplifies threading considerations, as multiple threads can look up a given texture and reason about its size/format/… without locking access.

Lifetime

The current mailbox lifetime is tied to the lifetime of the underlying service-side texture, which is ref-counted by all the client ids. In practice, producer contexts generally need to keep alive their texture while any consumer is active. There are some cases where we may reap the last active reference (e.g. when the producer process dies) before a consumer has a chance to take a reference, which causes issues (although rare).
With shared images, we need to either keep similar semantics (any client importing the image extends its lifetime) or define clear semantics if the shared image is destroyed while imported.

Design decisions

Client-side / API

We want to separate the conceptual shared image operations from the actual API, which may combine one or more operation together with other client API operations.

Shared image operations

- Explicit creation of images
    - 2 versions:
        - Create with format/size: Replaces **`glTexImage2D`**/**`glTexStorage2D`**/**`glTexStorage2DImageCHROMIUM`**

        - Create from GMB: Replaces **glCreateImageCHROMIUM** + **glBindTexImage2DCHROMIUM**
    - Usage flags (e.g. GL, Raster, Video, Display)
        - TODO: do we need to know at this time if the resource will be concurrently used?
    - Gives out the identifier (mailbox)

  This allows us to allocate the memory using the right API based on which driver API we need to interact with.

- Explicit import into client API: **GLES2Interface**/**RasterInterface**/…
    - Replaces **CreateAndConsumeTextureCHROMIUM** and **glBindTexImage2DCHROMIUM**
- This allows amortizing the cost of creating the per-driver API objects/handles. The image can be imported concurrently in multiple client contexts, but can't be used outside of a **Begin**/**End** scope (see below).

- Explicit **Begin**/**End**
    - With usage modes: **READ_WRITE** (Exclusive) / **READ_ONLY** (Shared)
        - Write accesses need **READ_WRITE** mode in the client context.
        - Read accesses need **READ_ONLY** or **READ_WRITE**.
        - **Begin**(**READ_WRITE**) only succeeds if all other context have ended their scope
        - **Begin**(**READ_ONLY**) only succeeds if no other context is in a **READ_WRITE** scope.
        - Invalid overlapping scopes cause command buffer errors (e.g. **GL_INVALID_OPERATION**) or lost context.
    - New concept, though also implicit in current **glBindTexImage2DCHROMIUM** / **glReleaseTexImage2DCHROMIUM**. Will need to be added in client-side code.
        - Can be folded into **BeginRaster**/**EndRaster**, **CopySubTexture**
    - Does not supersede sync tokens
        - Sync tokens deal with cross-stream command ordering
        - **Begin**/**End** deal with service-side synchronization within an ordered stream of commands
        - But could add wrappers for **WaitSyncToken**+**Begin** / **End**+**GenSyncToken** if that's helpful.
    - Multi-Begin/End form? To avoid synchronization overhead, e.g. for Display.

  This allows a concrete place to deal with transitions (layout/queue), synchronization, and interop in general.

Client API

This is what the end state of the client APIs should look like. See below for what we should do during the transition.

Creation API: **SharedImageInterface**

New interface that allows explicit creation/deletion of shared images.

```
Mailbox CreateSharedImage(viz::ResourceFormat format,
                          const gfx::Size& size,
                          const gfx::ColorSpace& color_space,
                          uint32_t usage)
Mailbox CreateSharedImage(GpuMemoryBuffer* gmb,
                          const gfx::ColorSpace& color_space,
                          uint32_t usage)
SyncToken GenUnverifiedSyncToken()
void VerifySyncTokens(base::span<SyncToken>)
void DestroySharedImage(const SyncToken& sync_token,
                        const Mailbox& mailbox)
```

This interface isn't stateful, doesn't need to be thread-bound like e.g. **GLES2Interface**, and could be implemented directly by **GpuChannelHost**, on its own separate stream.

Note: we could also add the same APIs to ContextSupport

GLES2Interface:

Remove:

```
GenMailboxCHROMIUM
ProduceTextureDirectCHROMIUM
CreateAndConsumeTextureCHROMIUM
CreateImageCHROMIUM
DestroyImageCHROMIUM
BindTexImage2DCHROMIUM
BindTexImage2DWithInternalformatCHROMIUM
ReleaseTexImage2DCHROMIUM
TexStorage2DImageCHROMIUM
```

Add:

```
void CreateAndTexStorage2DSharedImageCHROMIUM(GLuint internalformat,
                                              GLbyte* mailbox)
void BeginSharedImageAccessDirectCHROMIUM(GLuint texture, GLenum mode)
void EndSharedImageAccessDirectCHROMIUM(GLuint texture);
```

RasterInterface:

Remove:

```
BindTexImage2DCHROMIUM
CreateAndConsumeTexture
CreateImageCHROMIUM
```

CreateTexture
DeleteTextures
DestroyImageCHROMIUM
GenMailbox
ProduceTextureDirect
ReleaseTexImage2DCHROMIUM
SetColorSpaceMetadata
TexParameteri
TexStorage2D
TexStorage2DImageCHROMIUM

Change:
CompressedCopyTextureCHROMIUM and **CopySubTexture** take mailbox arguments and do an implicit Begin/End on the source (Shared mode) and destination (Exclusive mode), if they are backed by a shared image.
BeginRasterCHROMIUM takes a mailbox argument and does an implicit Begin with Exclusive mode
EndRasterCHROMIUM does an implicit End

Implementation (service-side)

- Multiple types of **gpu::SharedImageBacking**, 1 per image, chosen depending on usage flags at image creation.
    - A. GL texture (only GL<->GL sharing, legacy fallback, e.g. Linux without vulkan) with share groups. Synchronization implicit (legacy) or using GL sync object
    - B. Vulkan memory + image (only Vulkan <-> Vulkan sharing, e.g. raster tiles / display), shared across contexts. Synchronization using Vulkan pipeline+memory barriers
    - C. Vulkan shareable memory opaque fd/handle (Vulkan <-> GL sharing, e.g. WebGL back buffer). Synchronization using opaque fd/handle semaphores (or more specific type if available).
    - D. Native GMB handle: **IOSurface**/dmabuf/**AHardwareBuffer**/DXGI shared resource (scanout <-> Vulkan and/or GL and/or other API sharing, e.g. video, overlays). Synchronization using platform-specific primitive: **glFlush** on Mac, kernel sync objects on Linux & Android, keyed mutex on Windows
    - E. An **EGLImage** backed by a GL texture, allowing sharing between GL contexts in different share groups (WebView).
    - F. (Optional). A **SkImage** shared between **RasterDecoder** and **SkiaRenderer**/**SkiaOutputSurfaceImpl**, for prototyping OOP-R+OOP-D.
- **gpu::SharedImageManager** provides a mapping from **gpu::Mailbox** to **gpu::SharedImageBacking**

- Multiple **`gpu::SharedImageRepresentation`**, possibly concurrent. Each context/user uses one given representation. Created on demand, possibly shared across contexts using the same type, when imported into client API.
    - GL texture, shared (#A), imported from opaque fd/handle (#C), imported from native GMB/**`GLImage`** (#D, #E)
    - Vulkan memory+image, either Vulkan-allocated (#B, #C), or imported (#D)
    - GMB (#D)
- Begin/End operate on the representation, but update state on the backing:
    - Current usage state: exclusive (write) / read / shared (multiple readers) / unused
    - Current driver-level state: GL texture params, Vulkan image layout and queue state (including **`EXTERNAL`**/**`FOREIGN`**)
    - Synchronization state: e.g. GL sync object/semaphore/keyed mutex value from the latest use to wait for.
- Threading:
    - **`gpu::SharedImageManager`** is thread-safe, maintains a mailbox->**`SharedImageBacking`** map, accessed under (short-lived) lock.
    - Each **`gpu::SharedImageBacking`** that may be accessed from multiple threads maintains an internal lock taken when operating on the internal state (from the representation).

Client code updates / migration

Producers

Producers need to change from allocating a GL texture and sharing it via a mailbox to allocating a shared image, and (if necessary) importing it into **`GLES2Interface`**/**`RasterInterface`**, and scoping access between Begin/End.

Exhaustive list (as of 2018-05-09) of non-test producers:

- Compositor
    - HUD layer: uses **`TextureAllocation`** to create a texture, its storage, then export to a mailbox. Change **`TextureAllocation`** to deal with shared images instead (explicit image allocation + import). Scope fairly trivial (within the same function).
    - Raster
        - GPU: uses **`TextureAllocation`**, but separates texture id creation and mailbox (on compositor thread/context) from storage (on raster worker). This is due to lifetime issues + **`GLES2Interface`**/**`RasterInterface`**

thread affinity, that should disappear with shared images, should be able to create on a worker thread. See the consumer for scope.
        - One-copy: same as GPU.
        - Zero-copy: texture/mailbox creation next to **GLImage** creation. Change to create a shared image out of GMB (likely no import to GL needed).
    - UI Resources: uses **TextureAllocation** to create texture/storage & export. Similar to the HUD layer.
- Media
    - **VideoResourceUpdater**: Produced right after texture creation+allocation (here) via **TextureAllocation**. Trivial to convert to explicit allocation+import. Usage scope:
        - **VideoResourceUpdater::CopyHardwarePlane**: trivial scope for copy (within a function).
        - **VideoResourceUpdater::CreateForSoftwarePlanes** (1, 2): 2 trivial scopes for upload
    - **GpuVideoAcceleratorFactoriesImpl**: Used to allocate textures for HW video decoder. Creates GL textures, possibly their storage too, and then produces to mailbox, and passes texture to video decoder. Decoder may use existing storage (in very limited cases), or replace storage under the hood. Will require major changes.
    - **PaintCanvasVideoRenderer**: produce a skia image that's the conversion from YUV/NV12 to RGB. Texture allocated inside of skia. Probably shouldn't use skia directly (needs gl), migrate to Resource/RasterInterface?
    - **VideoOverlayFactory::Texture**: creates **SCANOUT** GMB+image+GL texture, then produce. Change to explicit image allocation (import likely unnecessary).
    - **GpuMemoryBufferVideoFramePool**: creates **SCANOUT_CPU_READ_WRITE** GMB+image+GL texture, and produce. Change to explicit image allocation from GMB + import.
    - **GLES2DecoderHelper**: new decoder path (mojo). Service side, similar to **GpuVideoAcceleratorFactoriesImpl**, but GL textures never exposed to client side, should be easier to port.
    - **D3D11PictureBuffer**: Service side. related to the above.
    - StreamTextureFactory: Android-specific, old pre-AVDA path, but still used on blacklisted devices and for HLS (won't go away). GLImage under the hood with special IPC to update, can likely be expressed in terms of shared images, but exact details TBD
    - **MakeTextFrameForCast** for **WebMediaPlayerCast**: Android specific, create+allocate+upload+produce in a single function, should be trivial to change to explicit allocation + import + upload.
- WebGL Backbuffer: creates GL texture (with possibly **SCANOUT** GMB+image), then (lazily) produces. Uses as a backbuffer or a copy target. Change to explicit image

allocation + import. Scope can begin in **`CreateOrRecycleColorBuffer`** and end in **`FinishPrepareTransferableResourceGpu`**.
    - Note: also produced here or here for export to other WebGL/Canvas. Not sure there's a specific reason for it, we should just be able to use a single mailbox for this resource.
    - Also here for WebXR.
- Canvas Backbuffer
    - **`CanvasResourceGpuMemoryBuffer`**: Creates **`SCANOUT`** GMB+image+GL texture then produce, uses this to copy from canvas backbuffer. Change to explicit image allocation (consider merging with **`CanvasResourceProvider_Texture`** option after rework).
    - **`CanvasResource_Bitmap`** for **`CanvasResourceProvider_Texture`**: Complex one. At the end of the day, is uses **`GraphicsContext3DUtils::GetMailboxForSkImage`** to produce a mailbox out of a SkImage backed by a GL texture. The GL texture is allocated inside of Skia, **`CanvasResourceProvider`** takes a snapshot of the **`SkImage`** which triggers copy-on-write inside of Skia. This would need to be replaced by explicit image allocation (and import) on the chrome side. In the shorter term we could consider keeping CoW behavior and expose the new mailbox semantics to Skia, however if we want to move Canvas to **`RasterInterface`**, we'll need explicit allocations anyway.
- Display readback
        - mailbox on result of **`CopyTexImage2D`**. Change to explicit allocation + **`CopyTexSubImage2D`** (or equivalent).
- **`GLHelper`** clients
    - **`Reflector`** via **`OwnedMailbox`**. Maybe change in viz anyway (bug)? Otherwise, need to couple mailbox with texture creation, currently implicit on swap buffers with full damage. 2 cases:
        - **`OffscreenBrowserCompositorOutputSurface`**: mailbox created right before texture is allocated, easy to change order. Use scope is somewhat unclear unfortunately, but should be limited to between **`BindFramebuffer`** and **`SwapBuffers`**. Synchronization is somewhat iffy, but the reader (another DisplayCompositor) should be on the same stream, so no interleaving reads should happen.
    - **`GpuBrowserCompositorOutputSurface`**: mailbox creation separated from allocation (in **`SwapBuffers`**), and in particular, not only unallocated textures are produced (and likely used), but existing produced textures are resized, violating immutability. However it should be possible to change a bit to delay production until the first frame, and recreate when the size changes. On the plus side, the use scope is well defined, as a copy inside of **`SwapBuffers`** - modulo synchronization which again is somewhat iffy (but similarly, the reader should be on the same stream).

    - **CanvasCaptureHandler::ReadYUVPixelsAsync** (media capture from element). produces **SkImage**. **SkImage** comes from Canvas element (2D Canvas, WebGL backbuffer? To confirm.) Will need to pass mailbox instead of (on top of?) SkImage.
- **FastInk**: texture/mailbox creation next to **GLImage** creation. Change to create shared image out of GMB + import to GL. Switch to **RasterInterface** should be easy too.
- **Exo**: texture/mailbox creation decoupled from bind image, but should be fixable? Uses GL (for e.g. queries), maybe migrate to **ResourceInterface**?
- Pepper
    - 2d upload: Allocate textures + storage, then produces. Change to explicit image allocation + import.
    - 3d: Service side, Allocate texture + storage, then produces. Change to explicit (service-side) allocation
    - Video decoder (**PPB_VideoDecoder** for ARC, not Flash): Allocate textures (+ possibly storage), then produces. Change to explicit image allocation + import sounds appealing, however this is passed to the hardware video decoder, so same exact concern as **GpuVideoAcceleratorFactoriesImpl** above. A solution for it is likely portable to this use case. Another option is to port it to move to **MojoVideoDecoder** which should avoid all these issues.
    - Compositor layer: This is a big bummer. Texture is allocated by the pepper module (outside of the chrome tree), and there's no hope to change callers. This interface is used for (old-style) ARC.
- Android VR: texture allocated, produced. Later on, AHB-backed GMB allocated and bound to same texture, may happen multiple times (resize). This violates immutability, however the users don't rely on stability of the texture/mailbox, so it should be fine to change to explicit (re)create from GMB on resize (might even be able to reuse the mailbox - why not).
- Other **GraphicsContext3DUtils::GetMailboxForSkImage** clients (canvas?): They reduce to users of **AcceleratedStaticBitmapImage::CreateFromSkImage**, which are either:
    - **UnacceleratedStaticBitmapImage::MakeAccelerated**: uploads a CPU-side **SkImage** into a texture. Replace into explicit image allocation + import + upload
    - **StaticBitmapImage::Create** with a non-null context provider and a texture-backed image. It doesn't look like there is such a caller (some callsites call with a context provider, but AFAICT only pass in a bitmap image)

Consumers

Consumers need to add API calls to scope the accesses to the shared image.

- Display: Reader. Already existing scope (**Lock/UnlockForRead**).
- Media:
  - **VideoResourceUpdater**: Reader. Trivial scope for copy.
  - **PaintCanvasVideoDecoder** (1, 2, 3, 4): Reader. Trivial scope (within function) for copy.
- **GLHelper** clients
  - **ArcScreenCaptureSession**: Reader. Trivial scope for copy (Here)
  - **GLHelper** internals (1, 2): Reader, trivial scope for copy.
  - **Reflector**: this is a bit of a dud, provides a texture id in the client's context as opposed to the shared context (on the **OwnedMailbox**), but the shared context one is never used. There are some lifetime issues (the **OwnedMailbox** may need to outlive the client's context), but the shared context texture is otherwise never used. See producer side. At a high level, we should fix the lifetime issue at the shared image level.
- Pepper:
  - Video decoder (PPB_VideoDecoder for ARC, not Flash): Writer. if porting to **MojoVideoDecoder**, then problem disappear (MVD would be a producer). If keeping with a GVDAH hookup, scope would be while decoding, from **AssignPictureBuffers** or **ReusePictureBuffer** to **PictureReady** or **DismissPicture**. Could be implicit in GVDAH API
- **AcceleratedStaticBitmapImage::CopyToTexture** (Canvas/WebGL?): Reader. Trivial scope for copy.
- WebGL's DrawingBuffer::TransferToStaticBitmapImage: Reader. Non-trivial scope, texture goes into **SkImage** which goes into **AcceleratedStaticBitmapImage**. **AcceleratedStaticBitmapImage** can create new mailboxes out of the SkImage and/or new SkImages out of the mailboxes (ad lib). **AcceleratedStaticBitmapImage** should probably deal with the scope, but becomes difficult when we end up with a **SkImage** and pass that to Skia (refcounting, CoW, etc.).
- **DrawingBuffer::CopyToPlatformTexture** (WebGL): Reader. Trivial scope for copy.
- XR (VR/AR):
  - Back buffer (**XRWebGLDrawingBuffer**):
    - **OverwriteColorBufferFromMailboxTexture:** Reader. Trivial scope for copy (note: missing deletion of texture?)
    - **UseSharedBuffer**: Writer (GL FBO). Scope ends on XRWebGLDrawingBuffer::DoneWithSharedBuffer at the latest. TODO: is it the exact right scope?

        - **TransferToStaticBitmapImage** and **MailboxReleasedToMirror**: similar to WebGL's version, goes into **AcceleratedStaticBitmapImage** etc.
    - **GpuMemoryBufferImageCopy::CopyImage**: Reader. Trivial scope for copy
- **SkiaTextureHolder** (WebGL/Canvas): used for **AcceleratedStaticBitmapImage**, see above.
- Compositor:
    - GPU Raster and OOP Raster: Writer. Trivial scope. Note: it allocates here currently. See producer-side comments
    - One Copy: Writer. Trivial scope. ditto

Transition plan

These are large fundamental changes to many components in Chrome. It would be unreasonable to change everything at once, so we need a concrete transition plan to incrementally move to the new API/concepts.

There are 2 realistic options:

1. Implement new API on top of old one, which is fairly trivial (explicit create == **GenTextures**+**TexStorage**+**ProduceTexture**, import == **CreateAndConsumeTexture**, begin/end = noop). Steps:
    a. Add new API, emulated on top of old one, client-side
    b. Port all clients to new API
    c. Implement new API service-side
    d. Remove old API

    Pros: less trashing / throw-away code. Better confidence we won't get stuck in limbo where we can't get rid of the fallback path
    Cons: step (c) can't happen before step (b), we get burned by the long pole (e.g. may need deprecation/removal of some Pepper APIs before we can proceed).
2. Implement old API on top of new one with a fallback path
    a. Add new API, including a way to create a shared image out of an existing GL texture with restrictions lifted (e.g. begin/end not needed)
    b. Implement service-side with GL texture fallback path
    c. Implement old API on top of new API
    d. Port all clients to new API
    e. Implement other versions of new API's service side (e.g. pure Vulkan, GL interop, GMBs)
    f. Remove old API
    g. Remove fallback path.

Pros: Steps c, d and e can all happen in parallel. May let us test interop between some components (e.g. Raster + Display) before everything is ported over.
Cons: more trashing, more risk (e.g. lifted restrictions means the API usage may be inconsistent).

Decision: we chose option 2.