True PPGTT [part 3]

Author : Ben Widawsky

Contents

The Per-Process Graphics Translation Tables provide real process isolation among the various graphics processes running within an i915 based system. When in use, the combination of the PPGTT and the Hardware Context provide the equivalent of the traditional CPU process. Most of the same capabilities can be provided, and most of the same limitations come with it. True PPGTT encompasses all of the functionality currently merged into the i915 kernel driver that support page tables and address spaces. It's called, "true" because the Aliasing PPGTT was introduced first and often was simply called, "PPGTT."

The True PPGTT patches represent one of the more challenging aspects of working on a project like the Linux kernel. The feature couldn't realistically be enabled in isolation of the existing driver. When regressions occur it's likely that the user gets no display. To say we get chided on occasion would be an understatement. Ipso facto, this feature is not enabled by default. There are quite a few patches on the mailing list that build new functionality on top of this support, and to help stabilize existing support. If one wishes to try enabling the real PPGTT, one must simply use the i915 module parameter: **enable_ppgtt=2**. I highly recommended that the stability patches be used unless you're reading this in some future where the stability problems are fixed upstream.

Unlike the previous posts where I tried to emphasize the hardware architecture for this feature, the following will not go into almost no detail about how hardware works. There won't be PRM references, or hardware state machines. All of those mechanics have been described in parts 1 and part 2

EDIT1: I forgot to include a diagram I did of the software state machine for some presentation. I long lost the SVG, and it got kind of messed up, but it’s there at the bottom.

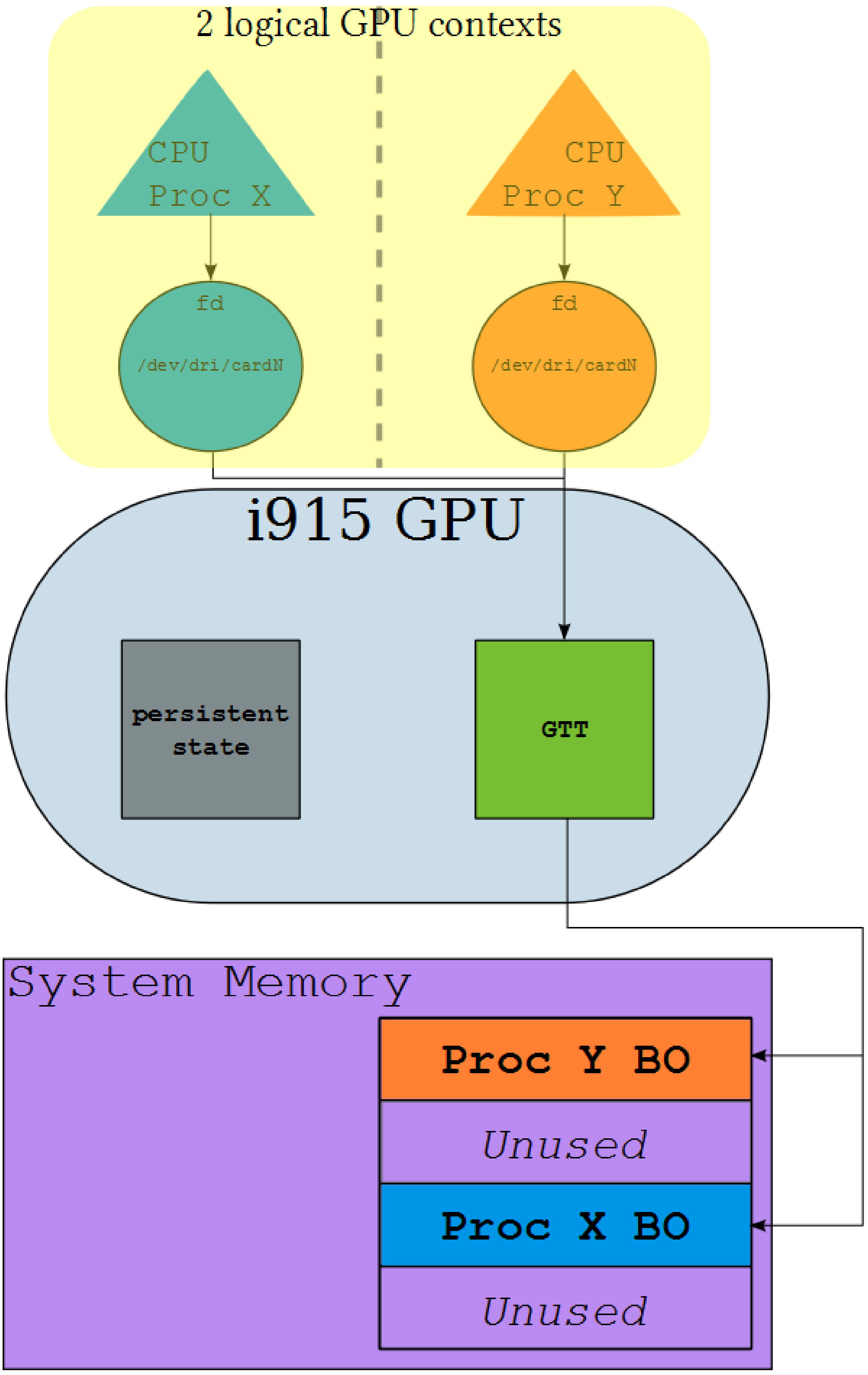
A Brief History of the i915 Graphics Process

There have been three stages of the definition of a graphics process within the i915 driver. I believe that by explaining the stages one can get a better appreciation for the capabilities. In the following pictures there is meant to be a highlighted region (yellow in the first two, yellow, orange and blue in the last) that denotes the definition of a GPU context/process with that feature. The definition of a process begins to bleed between CPU, and GPU, and indeed that is how it should be.

Unfortunately I have some overlap with my earlier post about Hardware Contexts. I found no good way to write this post without doing so. If you read that post, consider this a refresher.

File Descriptors

Initially all GPU state was shared by every GPU clients. The only partition was done via the operating system. Every process that does indirect rendering will get a file descriptor for the device. The file descriptor is the thing through which commands are submitted. This could be used by the i915 driver to help disambiguate “who” was doing “what.” This allowed the i915 kernel driver to prevent one GPU client from directly referencing buffers by making the buffer object handles per file descriptor. This is very easy to implement, it’s just an idr in the kernel. For applications which do not require context saved, non-buggy apps, or non-malicious apps, it is *still* perfectly sufficient. BO handles are just integers. The integers exist within the domain of the process. BO #1 for the X server is not the same as BO #1 for xonotic[1]. Even though we had this partition at the software level, nothing was enforced by the hardware. Provided a GPU client could guess where another buffer resided, it could easily operate on that buffer. Similarly, a GPU client could not expect the GPU state it had set previously to be preserved for any amount of time.


2 logical GPU contexts
CPU
Proc X
CPU
Proc Y
fd
/dev/dri/cardN
fd
/dev/dri/cardN
i915 GPU
persistent state
GTT
System Memory
Proc Y BO
Unused
Proc X BO
Unused

File descriptor isolation.Before hardware contexts.

Hardware Contexts

The next step towards isolation was the Hardware Context[2]. The hardware contexts built upon the isolation provided by the original file descriptor mechanism. The hardware context was an opt-in interface which meant that those not wishing to use the interface received the old behavior: they could purposefully or accidentally use the state from another GPU client[3]. There was quite a bit of discussion around this at the time the patches were in review, and there's not really any point in lamenting about it now.

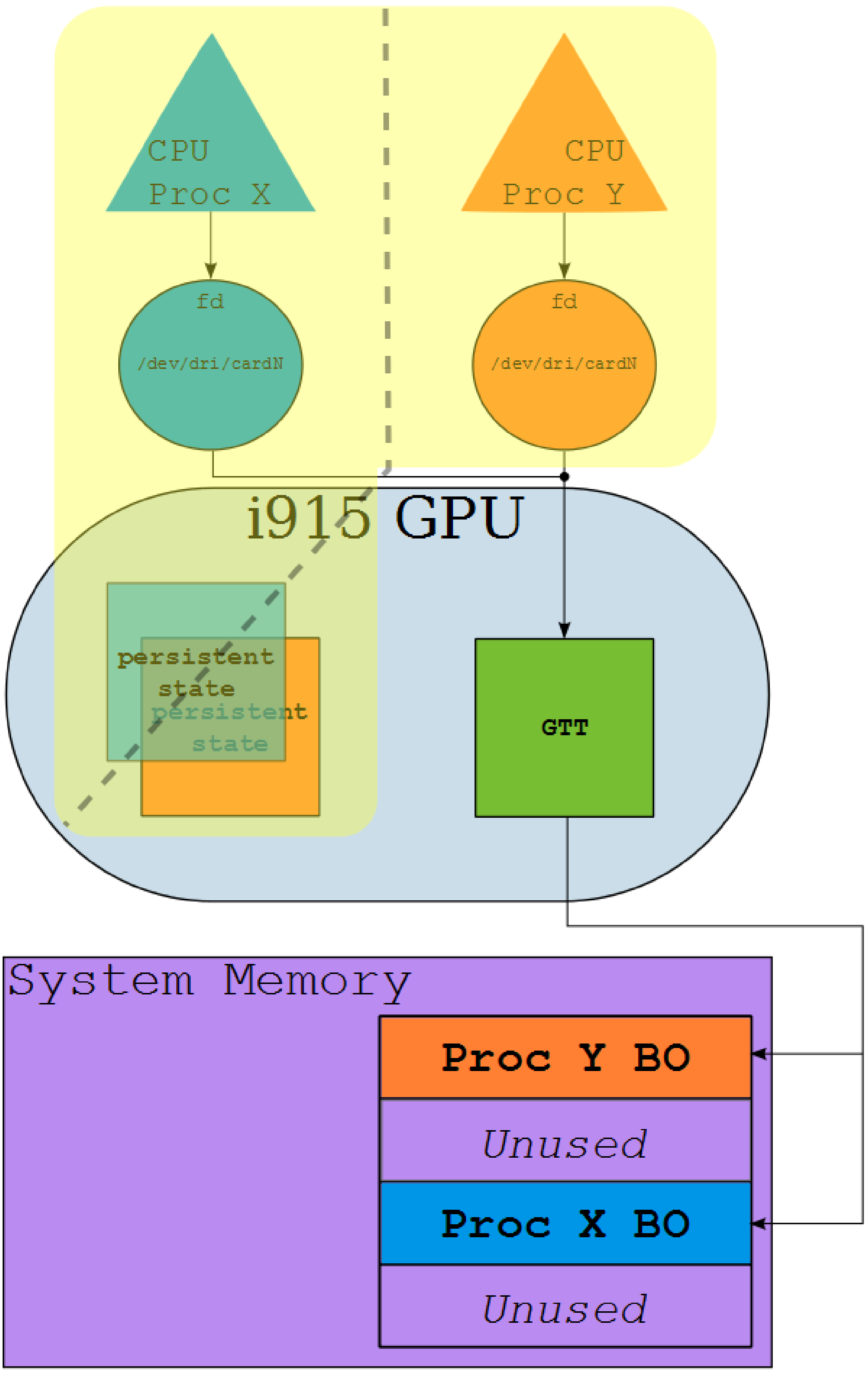
The context exists within the domain of the process/file descriptor in the same way that a BO exists in that domain. Contexts cannot be shared [intentionally]. The interface created was, and remains extremely simple.

```
  struct drm_i915_gem_context_create {   /* output: id of new context*
/   __u32 ctx_id;   __u32 pad;  };    struct drm_i915_gem_context_dest
roy {   __u32 ctx_id;   __u32 pad;  };
```

As you can see from the two IOCTL payloads above, I wasn't lying about the simplicity. There was not a great deal of varying functionality, there just wasn't a lot to add. Destroy is an optional call because we have the file descriptor and can clean up if a process does not. The primary motivation for destroy() is simply to allow very meticulous and memory conscious GPU clients to keep things nice and clean. Earlier I had a list of 3 types of GPU clients that could survive without this. Well consider their inverse, this takes one of those off the list.

- ~~GPU clients needed HW context preserved~~
- Buggy applications writing to random memory
- Malicious applications

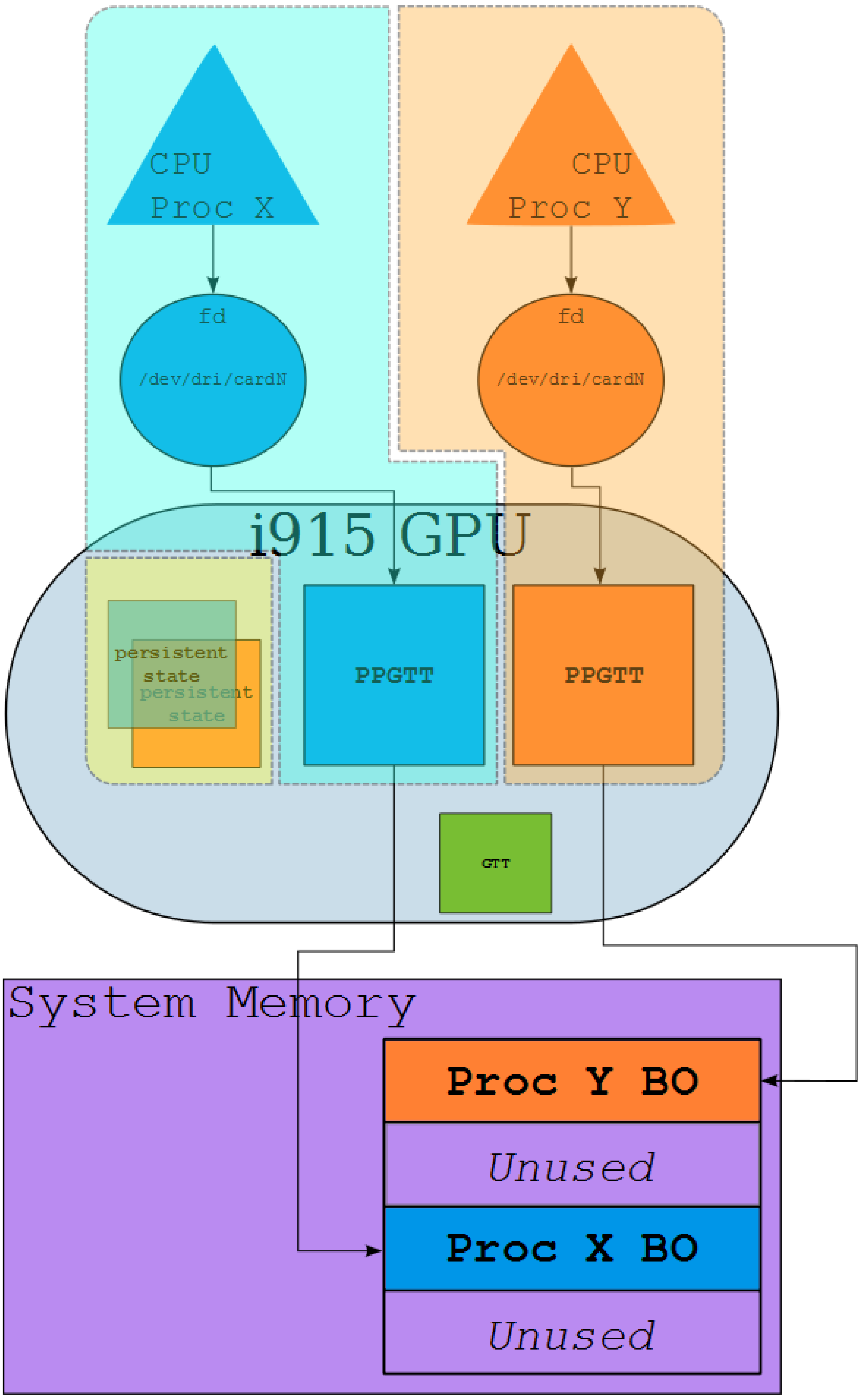
The block diagram is quite similar to above diagram with the exception that now there are discrete blocks for the persistent state. I was a bit lazy with the separation on this drawing. Hopefully, you get the idea.


CPU
Proc X
CPU
Proc Y
fd
/dev/dri/cardN
fd
/dev/dri/cardN
i915 GPU
persistent state
persistent state
GTT
System Memory
Proc Y BO
Unused
Proc X BO
Unused

Hardware context isolation

Full PPGTT

The last piece was to provide a discrete virtual address space for each GPU client. For completeness, I will provide the diagram, but by now you should already know what to expect.


CPU
Proc X
CPU
Proc Y
fd
/dev/dri/cardN
fd
/dev/dri/cardN
i915 GPU
persistent
state
persistent
state
PPGTT
PPGTT
GTT
System Memory
Proc Y BO
Unused
Proc X BO
Unused

PPGTT, full isolation

If I write about this picture, there would be no point in continuing with an organized blog post :-). So I'll continue to explain this topic. Take my word for it that this addresses the other two types of GPU clients

- ~~*GPU clients needed HW context preserved*~~
- ~~Buggy applications writing to random memory~~
- ~~Malicious applications~~

Since the GGTT isn't really mentioned much in this post, I'd like to point out that the GTT still exists as you can see in this diagram. It is required for several components that were listed in my previous blog post.

VMAs and Address Spaces (AKA VMs)

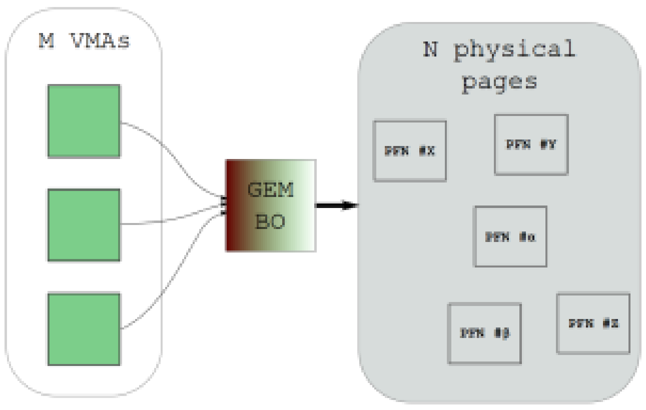
The patch series which began to implement PPGTT was actually a separate series. It was the one that introduced the **Virtual Memory Area** for the PPGTT, simply referred to as, VMA[4]. You can think of a VMA in a very similar way to a GEM BO. It is an identifiable, continuous range within an address space. Conceptually there isn't much difference between a GEM BO. To try to define it in my horrible math jargon: a logical grouping of virtual addresses representing an operand for some GPU operation within a given PPGTT domain. A VMA is uniquely identified via the tuple (BO, Address space). In the likely case that I made no sense just there, a VMA is just another handle on a chunk of GPU memory used for rendering.

Sharing VMAs

You can't (see the note at the bottom). There's not a whole lot I can say without doing another post about DMA-Buf, and/or Flink. Perhaps someday I will, but for now I'll keep things general and brief.

It is impossible to share a VMA. To repeat, a VMA is uniquely identifiable by the address space, and a BO. **It remains possible to share a BO.** An address space exists for an individual GPU client's process. Therefore it makes no sense to share a VMA since the address space cannot be shared[5]. As a result of using the existing sharing interfaces a GPU will get multiple VMAs that reference the same BO. Trying to go back to the math jargon again:

1. VMA: (BO, Address Space) // Some BO mapped by the address space.
2. VMA?: (BO?, Address Space) // Another BO mapped into the address space
3. VMA?: (BO, Address Space?) // The same BO as 1, mapped into a different address space.



M = {1,2,3,…} N = {1,2,3,…}

In case it’s still unclear, I’ll use an example (which is kind of a simplified/false demonstration). The scanout buffer is the thing which is displayed on the screen. When doing frontbuffer rendering, one directly renders to that buffer. If we remember my previous post, the Display Engine **requires** a GGTT mapping. Therefore we know we have VMA_{global}. Jumping ahead, a GPU client cannot have a global mapping, therefore, to render to the frontbuffer it too has a VMA, VMA_{pp}. There you have two VMAs pointing to the same Buffer Object.

NOTE: You can actually share VMAs if you are already sharing a Context/PPGTT. I can’t think of any real world examples off of the top of my head, but it is possible, and potentially a useful thing to do.

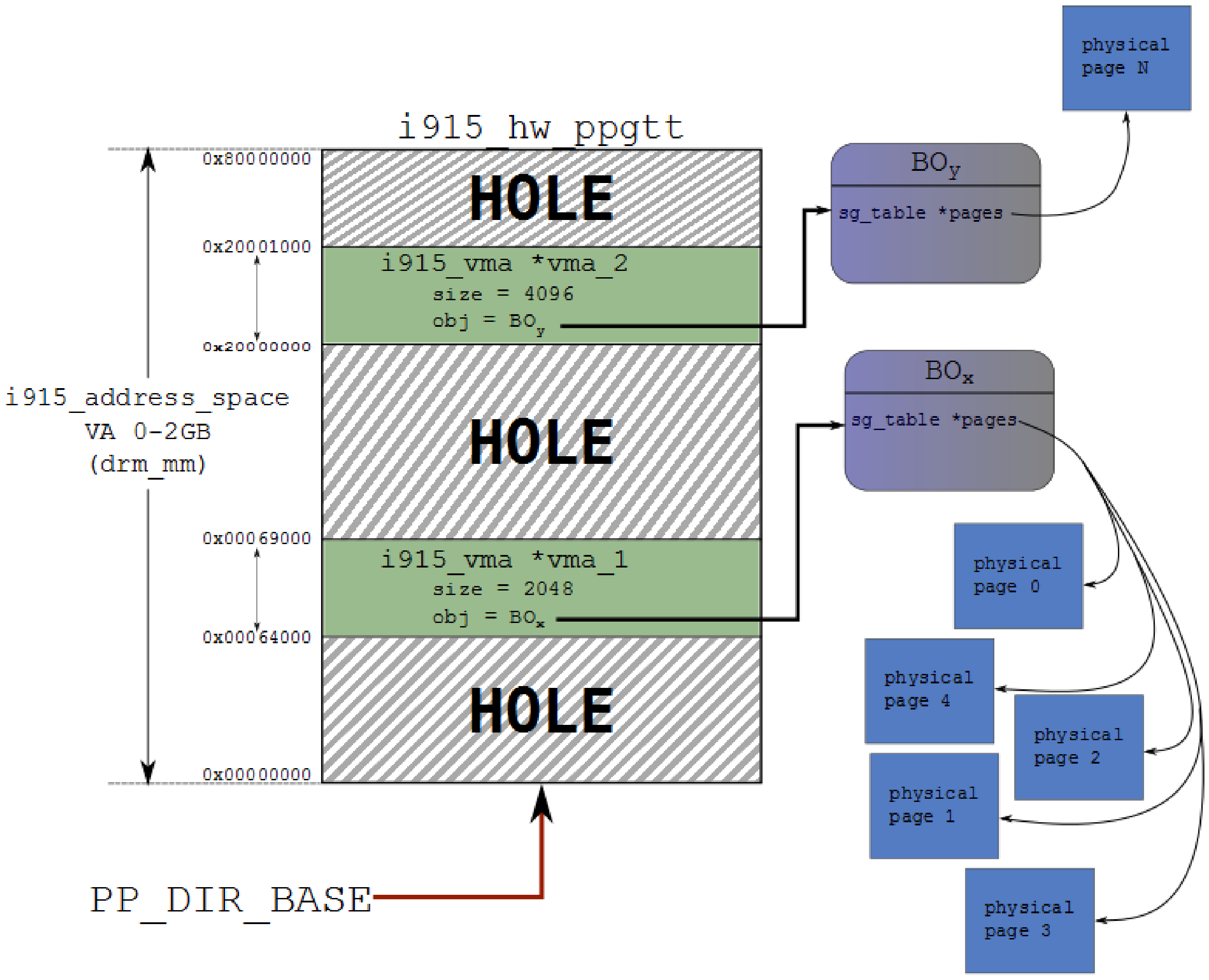
Data Structures

Here are the relevant data structures cropped for the sake of brevity.

```
  struct i915_address_space {          struct drm_mm mm;   unsigned lo
ng start;            /* Start offset always 0 for dri2 */   size_t tot
al;           /* size addr space maps (ex. 2GB for ggtt) */   struct l
ist_head active_list;   struct list_head inactive_list;  };    struct 
i915_hw_ppgtt {          struct i915_address_space base;   int (*switc
h_mm)(struct i915_hw_ppgtt *ppgtt,      struct intel_engine_cs *ring, 
     bool synchronous);    };  struct i915_vma {          struct drm_m
m_node node;          struct drm_i915_gem_object *obj;          struct
 i915_address_space *vm;  };
```

The ***struct i915_hw_ppgtt*** is a subclass of a ***struct i915_address_space***. Only two implementations of the super class exist: the i915_hw_ppgtt (a PPGTT), and the i915_gtt (the GGTT). It *might* make some sense to create a new PPGTT subclass for GEN8+ but I’ve not opted to do this. I feel there is too much duplication for not enough benefit.

I've already explained in different words that a range of used address space is the VMA. If the address space has the drm_mm, then it should make direct sense that the VMA has the drm_mm_node because this is the used part of the address space[6]. In the i915_vma struct above is a pointer to the address space for which the VMA exists, and the object the VMA is referencing. This provides the tuple that define the VMA.



HOLE 0×0->0×64000
VMA 1 0×64000->0×69000
HOLE 0×69000->512M
VMA 2 512M->512.004M
HOLE ~512M->2GB
Allocated space: 0×6000 Free space: 0x7fffa000

Relation to the Hardware Context

```
  struct intel_context {   struct kref ref;   int id;   ...   struct i
```

```
915_address_space *vm;  };
```

With the 3 elements discussed a few times already: file descriptor, context, PPGTT, we get real GPU process isolation. Since the context was historically an opt-in interface, changes needed to be made in order to keep the opt-in behavior yet provide isolation behind the scenes regardless of what the GPU client tried to do. If this was not done then innocent GPU clients could feel the wrath. Since the file descriptor was already intimately connected with the direct rendering process (one cannot render without getting a file descriptor), we can hook off of that to create the contexts and PPGTTs.

Implicit Context ("private default context")

From here on out we can consider a, "context" as the 3 elements: fd, HW context, and a PPGTT. In the driver as it exists today **if you do not provide a context for rendering, you cannot rely on GPU state being preserved.** A context is created for GPU clients that do not provide one, but the state of this context should be considered completely opaque to all GPU clients. I've called this the Private Default Context as it very much resembles the default context that exists for the whole system (again, let me point you to the previous blog post on contexts). The driver will isolate the various contexts within the system from implicit contexts, and vice versa. Hardware state may be corrupted for those using the private default context, where the hardware state will be preserved for those creating new contexts explicitly via the IOCTL.

The behavior of the implicit context does result in waste when userspace uses contexts (as Mesa does). There are a few solutions to this problem, and I've submitted patches for all of them (I can count 3 from the top of my head). *Perhaps one day in the not too distant future, this above section will be false and we can just say – every process will get a context when they open the DRI file. If they want more contexts, they can use the IOCTL.*

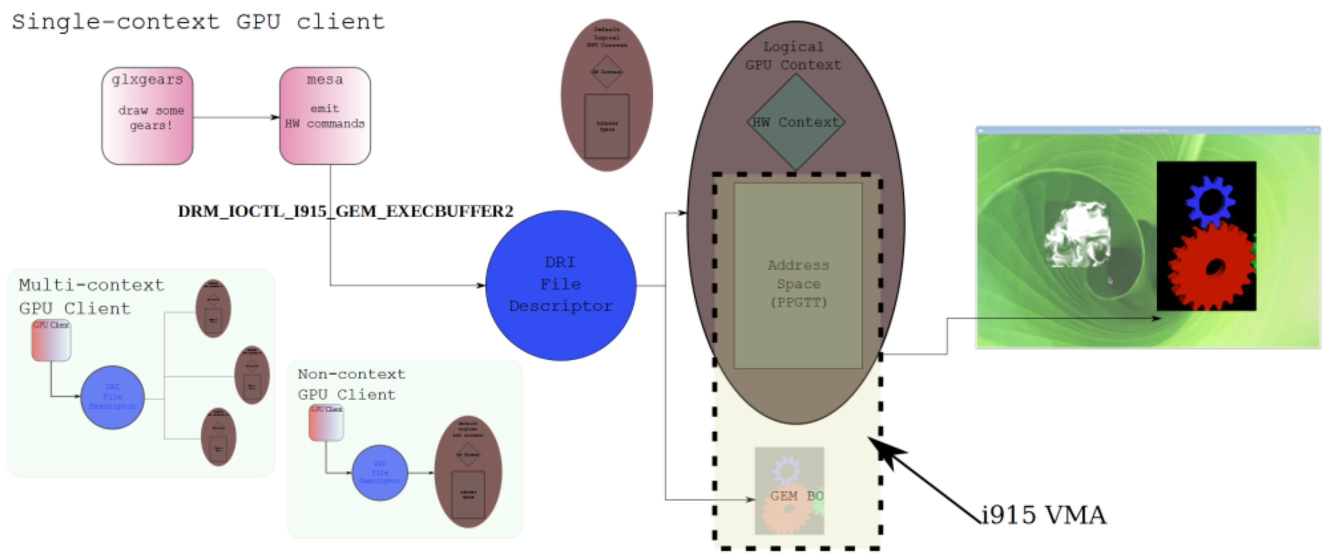
Multi Context

A GPU client can create more than one context. The context they wish to use for a given rendering command is built into the execbuffer2 API (not KMS is not context aware).

```
  struct drm_i915_gem_execbuffer2 {   /**     * List of gem_exec_object
2 structs    */   __u64 buffers_ptr;   __u32 buffer_count;     /** Off
set in the batchbuffer to start execution from. */   __u32 batch_start
_offset;   /** Bytes used in batchbuffer from batch_start_offset */
__u32 batch_len;   ...   __u64 flags;   __u64 rsvd1; /* now used for c
ontext info */   __u64 rsvd2;  };
```

This thing which initially seemed like a wart (from the HW context IOCTL, and inability to break ABI) early in PPGTT development maybe turned out to be useful. A process may wish to create several GL contexts. The API allows this, and for reasons I don't understand, it's something some applications wish to do. If there was no mechanism to create a new contexts, userspace would be forced to open a new file descriptor for each GL context. Or not reap the benefits of everything we've discussed for a GL context.

Image Overview



Overview

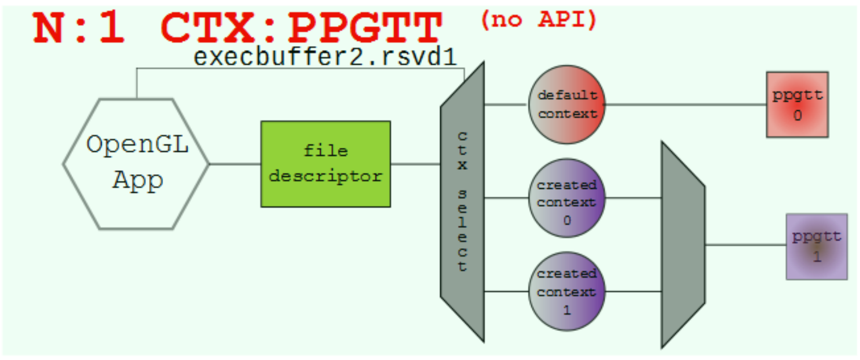
Context:PPGTT

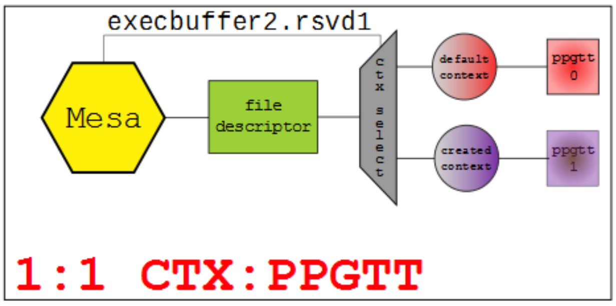
One of the more contentious topics in the very early stages of development was the relationship and connection of a PPGTT and a HW context.

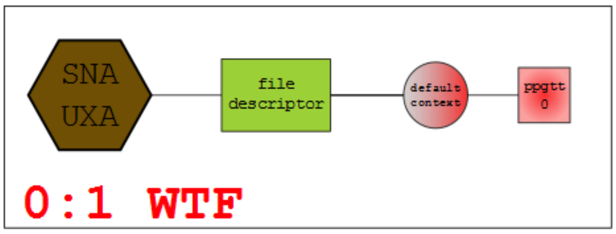
Quoting myself from one of earlier public declarations, here:

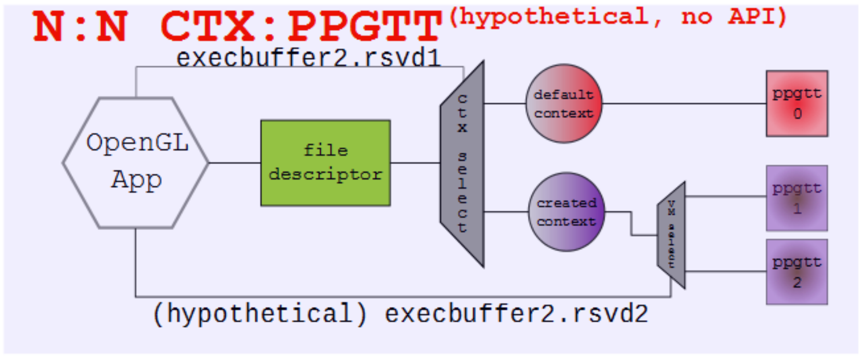
```
My long term vision is for contexts to have a 1:1 relationship with a
PPGTT. Sharing objects between address spaces would work similarly to
the flink/dmabuf model if needed.
```

My idea was to embed the PPGTT within the context structure, and creating a context always resulted in a new PPGTT. Creating a PPGTT by itself would have been impossible. This is **not** what we ended up doing. Behaviorally, it matches what we have today when using full PPGTT, and personally, I think it's a bit easier to treat it this way. The implementation allows multiple hardware contexts to share a PPGTT. I'm still unclear exactly what is needed to support share groups within OpenGL, but it has been speculated that this is a requirement for share groups. Fundamentally this would allow the client to create multiple GPU contexts that share an address space (it resembles what you'd get back when there was only HW contexts). The execbuffer2 IOCTL allows one to specify the context.





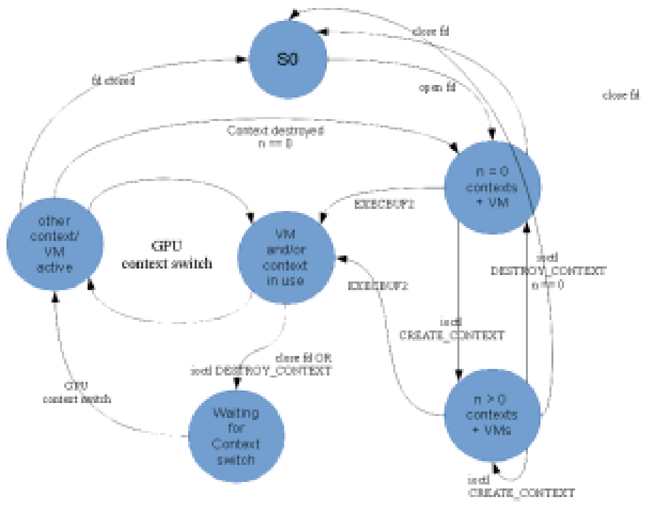




Current Mesa
Current DDX
2 hypothetical scenarios

Conclusion

I didn't actually review any of my images, so I hope they are right. Please feel free to send me issues, and I'll try to fix it as time allows[7]. Here is a state machine that I did for a presentation. Things got messed up, and I lost the SVG, but perhaps it will be of some value to someone.



State Machine

As I alluded to earlier, there is still some work left to do in order to get this feature turned on by default. I gave the links to some patches, and the parameter to make it happen. If you feel motivated to help get this stuff moving forward, test it, report bugs, try to fix stuff, don’t yell at me when things break :-).

That’s most of it. I like to give the 10 second summary.

1. i915_vma, i915_hw_ppgtt, i915_address_space: important things.
2. The GPU has a virtual address space per DRI file descriptor.
3. There is a connection between the PPGTT, and a Hardware Context.
4. VMAs are backed by BOs which are backed by physical pages.
5. GPU clients have some flexibility with how they interact with contexts, and therefore the PPGTT.

And finally, since I compared our now well defined notion of a GPU process to the traditional CPU process, I wanted to create a quick list of what I think are some interesting data points regarding the capabilities of the processors.

Thing	Modern X86 CPU	Modern i915 GPU
Phys Address Limit	48b?	~40b
Process Isolation	Yes	Yes (with True PPGTT)
Virtual Address Space	Yes	Yes
64b VA Space	Yes	GEN8+ 48b only
PTE access controls	Yes	No
Page Fault Handling	Yes	No
Preemption8	Yes	No

So while True PPGTT brings the GPU closer to having all of the [what I consider to be] interesting features of a modern x86 CPU – it still has a ways to go. I would be surprised if things **didn't** continue going in this direction.

SVG Links

As usual, please feel free to do something useful with the images I've created. Also as usual, they are really poorly named.
https://bwidawsk.net/blog/wp-content/uploads/2014/07/pre-context.svg
https://bwidawsk.net/blog/wp-content/uploads/2014/07/post-context.svg
https://bwidawsk.net/blog/wp-content/uploads/2014/07/post-ppgtt.svg
https://bwidawsk.net/blog/wp-content/uploads/2014/07/vma-bo-page.svg
https://bwidawsk.net/blog/wp-content/uploads/2014/07/vma.svg
https://bwidawsk.net/blog/wp-content/uploads/2014/07/ppgtt-context.svg
https://bwidawsk.net/blog/wp-content/uploads/2014/07/multi-context.svg

1. It’s technically possible to make them be the same BO through the two buffer sharing mechanisms. ?

2. Around the same time Hardware Contexts were introduced, so was the Aliasing PPGTT. The Aliasing PPGTT was interesting, however it does not contribute to any part of the GPU “process” ?

3. Hardware contexts use a mechanism which will inhibit the restoration of state when not opted-in. This means if one GPU client does opt-in, and another does not, the client without contexts can reuse the state of the client with contexts. As the address space is still shared, this is actually a really dangerous thing to allow. ?

4. I would have preferred the reservation of a space within the address space be called a, “GVMA”, but that was shot down during review ?

5. There’s a whole section below describing how this statement could be false. For now, let’s pretend address spaces can’t be shared ?

6. For those unfamiliar with the Direct Render Manager memory manager, a drm_mm is the structure for the memory manager provided by the DRM ~~midlayer~~ helper layer. It does all the things you’d expect out of a memory manager like, find free nodes, allocate nodes, free up nodes… A drm_mm_node is a structure representing an allocation from the memory manager. The PPGTT code relies entirely on thedrm_mm and the DRM helper functions in order to actually do the address space allocations and frees. ?

7. You have no idea how many people pointed out that I “mispelled” “defeca7e” in my last post ?

8. I am defining the word
   preemption as the ability to switch at an arbitrary point in time between contexts. On the CPU