AMD

THE WINDOWS KERNEL-MODE GRAPHICS DRIVER

SHIHCHEN CHEN
MODIFY BASE ON
ANTHONY BROWN

MAY 2018

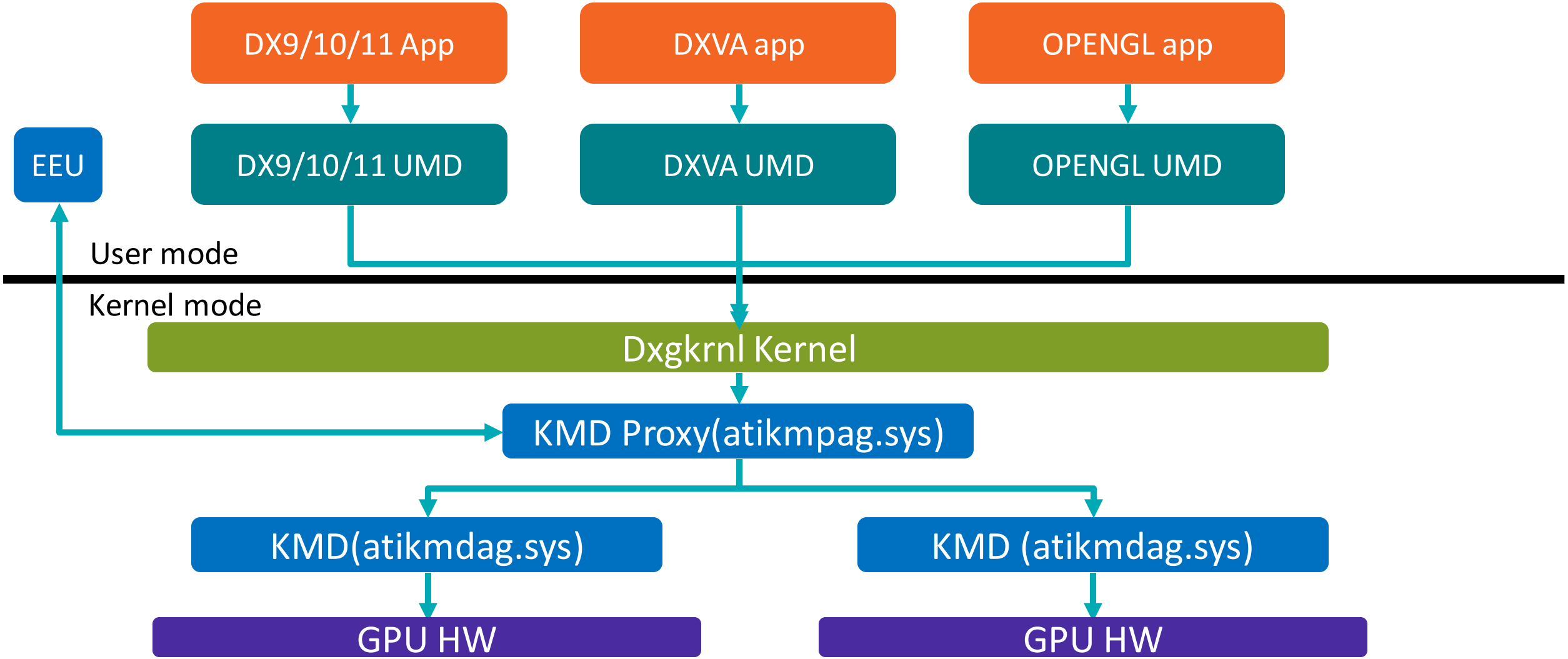
KMD TRAINING

AMD

- Windows KMD Overview
- Base Graphics Architecture
- EEU
- GPU Submissions
- Display and VIDPN
- Base Driver Features
- Multi-GPU and Proxy
- Interrupt
- TDR
- Flip/Present
- Engines, nodes, contexts, etc.
- Ring Buffers
- Memory in the KMD
  - Driver Memory
  - GPU Memory
  - Paging and residency
  - Virtual Addressing

WINDOWS KMD OVERVIEW

SOFTWARE STACK

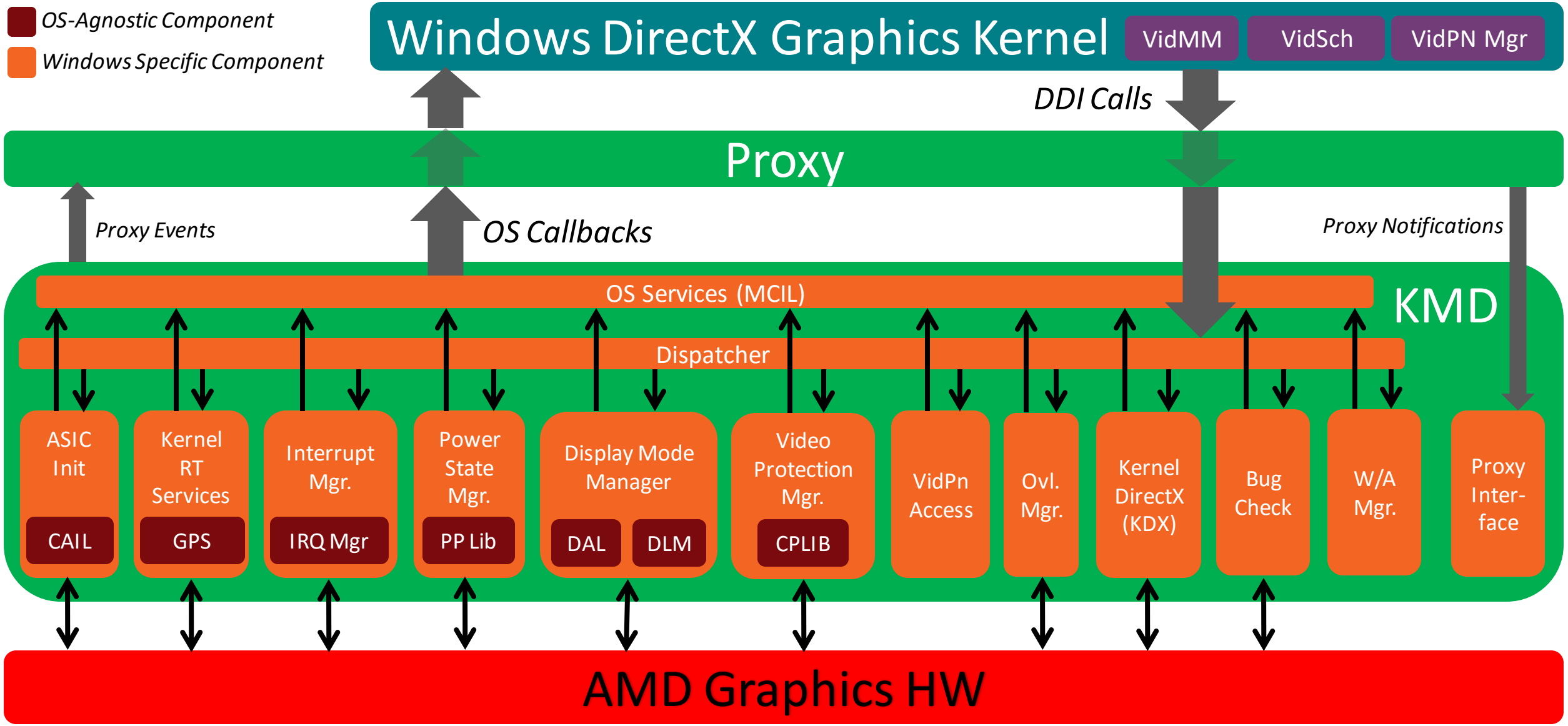


BASE GRAPHICS DRIVER STRUCTURE

AMD

- Two binaries in the Windows/System32/Drivers folder:
  - atikmpag.sys
    - **Kernel Mode Proxy Driver** (aka, KPD, Proxy)
    - Owned by CoreDev team
    - Always installed with driver, but operates in passthrough mode on single-GPU systems
    - Main purpose is to support multi-GPU features such as CrossFire, PowerXpress, MGPU SLS, Detachable Graphics, etc.
  - atikmdag.sys
    - **AMD Kernel Mode Driver** (aka KMD, base graphics driver, display driver, miniport driver)
    - Comprised of many subcomponents:
      - **Owned by CoreDev (KMD) team:** OsServices, Dispatcher, Display Mode Manager (DMM), DAL Link Manager (DLM), Kernel DirectX (KDX), Overlay Manager, Interrupt Manager, ASIC Initialization, Kernel Runtime Services (KRS), Power State Manager, etc.
      - **Owned by other teams:** DAL (Display), PPLib (Prime), CAIL/IRQMgr/GPS (CGI), CPLIB (MMD)
    - Windows-only, although many subcomponents are shared with other OS base drivers
- OS loads the Proxy Driver and the Proxy Driver loads the KMD(s)

KMD ARCHITECTURE



EEU

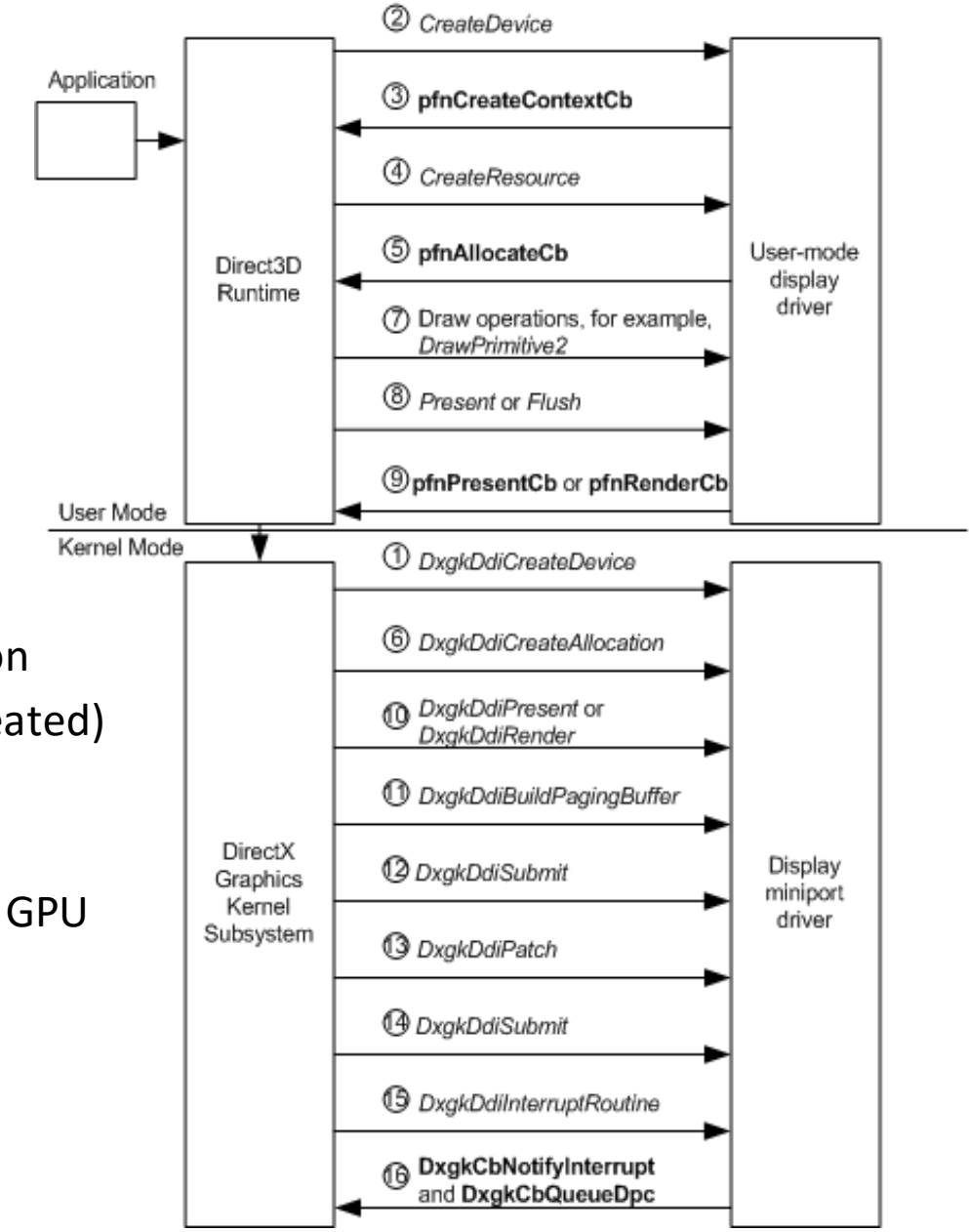
AMD

- The External Events Utility
- Atieclxx.exe (client) and atiesrxx.exe (service)
- User-Mode service/application installed with the graphics driver.
- Owned by CoreDev, but not part of the KMD
- Main purpose is to support the base driver in doing tasks that cannot be accomplished from Kernel Mode:
  - Change the topology or desktop resolution
  - Trigger display/mode re-enumeration
  - Respond to Windows Notification Messages
  - Trigger PnP Stop/Starts
  - Hooking into user-mode calls
  - Modify system power policy
- Sends messages to the KMD via escape calls
- Receives messages from the KMD through event objects and message queues
- Able to interact with CCC

GPU SUBMISSION

RENDER-RELATED DDIS

- Device/Context:
  - **Create/DestroyDevice:** Typically per-process
  - **Create/DestroyContext:** 1 or more per engine used by device
- Allocation Management:
  - **Create/DestroyAllocation:** Creates a description of a buffer
  - **Open/CloseAllocation:** Binds a buffer to a device
- Render:
  - **Render(KM/GDI):** Produce a DMA buffer to be scheduled for submission
  - **Patch:** Patch a given DMA buffer with physical addresses (semi-depreceated)
  - **Present:** Copy something to the primary surface (usually)
  - **BuildPagingBuffer:** Prepare a DMA paging buffer to be submitted
  - **SubmitCommand(Virtual)**: Submit a DMA buffer to be executed by the GPU
  - **PreemptCommand:** Preempt an in-flight DMA buffer



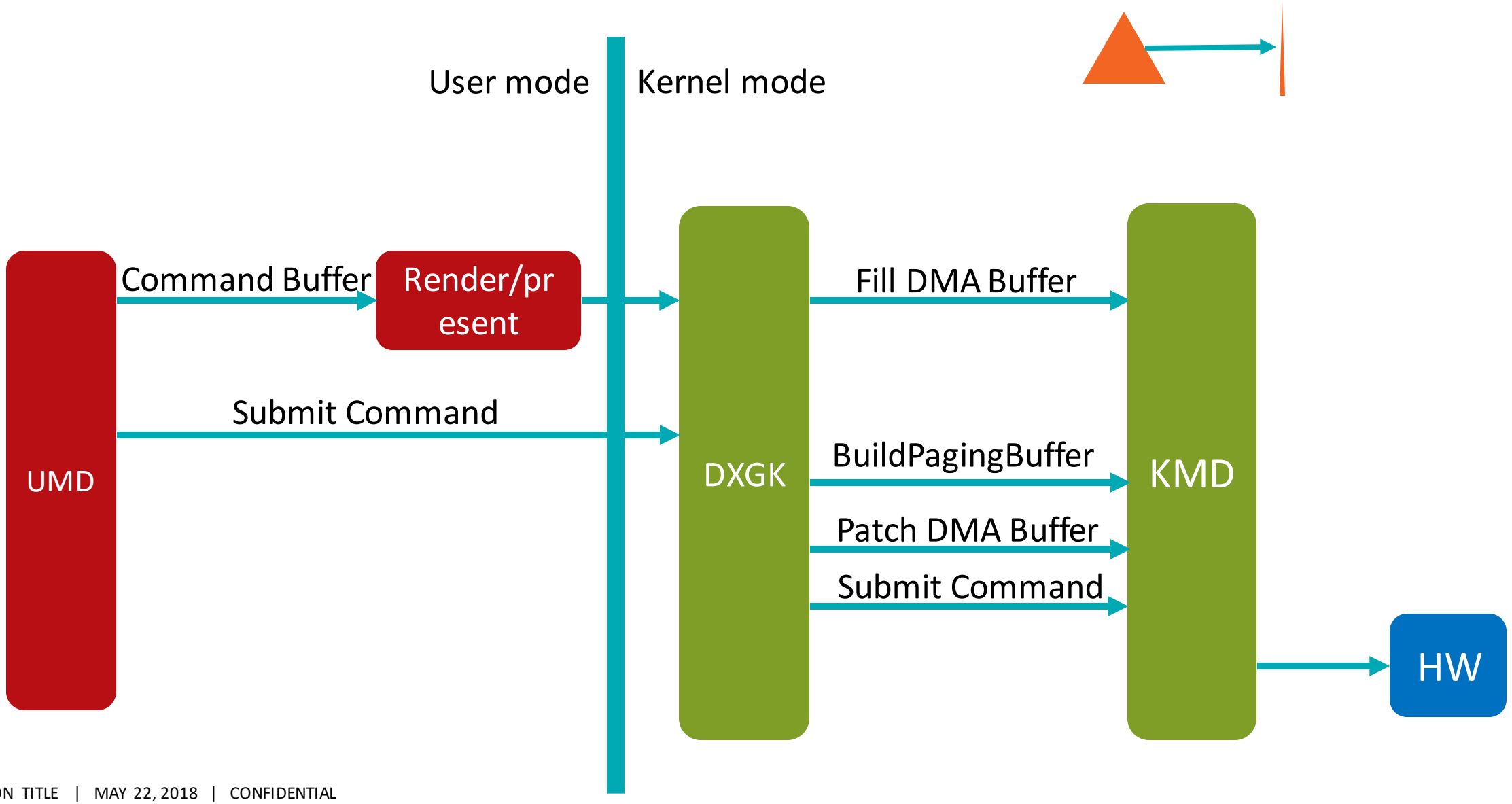
GPU SUBMISSIONS

DMA BUFFERS

- A **DMA buffer** is an area of memory containing a sequence of GPU instructions, called **PM4** (programming model 4)
- A DMA buffer is allocated by the OS or the UMD, typically in GPU-accessible system memory
- PM4 instructions are fetched by the GPU's CP (usually from system memory via DMA) and executed by a GPU engine
- Each DMA buffer has a unique identifier (**fence**) that's used to indicate its completion
- The method through which a DMA buffer is populated and submitted differs depending on the WDDM/driver model:
  - **The "Basic" WDDM 1.x model:**
    - UMDs populates an OS-provided user-mode **command buffer** with PM4 and submits it to the OS
    - The OS passes the command buffer to the KMD in DxgkDdiRender
    - KMD validates the contents of the command buffer and copies them to an OS-provided DMA buffer. KMD returns the DMA buffer to the OS.
    - The OS scheduler calls the KMD's DxgkDdiSubmitCommand DDI and passes in the DMA buffer to be submitted to the GPU by KMD
  - **The "Basic++" WDDM 1.x model:**
    - UMD allocates its own GPU-accessible **user-mode DMA buffer** (UDMA) and populates it with PM4. When work is submitted through the OS, the OS command buffer is essentially unused.
    - The OS passes the (empty) command buffer to the KMD in DxgkDdiRender, but KMD doesn't need to do any copy into the OS DMA Buffer.
    - When the OS scheduler calls DxgkDdiSubmitCommand, KMD submits the UDMA buffer to the GPU instead of the OS DMA buffer
  - **The WDDM 2.0 model:**
    - Essentially the same as the Basic++ model, but the OS supplies the "UDMA buffer" for the UMD (though that terminology isn't used)
    - Again, no copying necessary in the KMD
    - The command buffer the UMD fills with PM4 is the same buffer the GPU fetches commands from

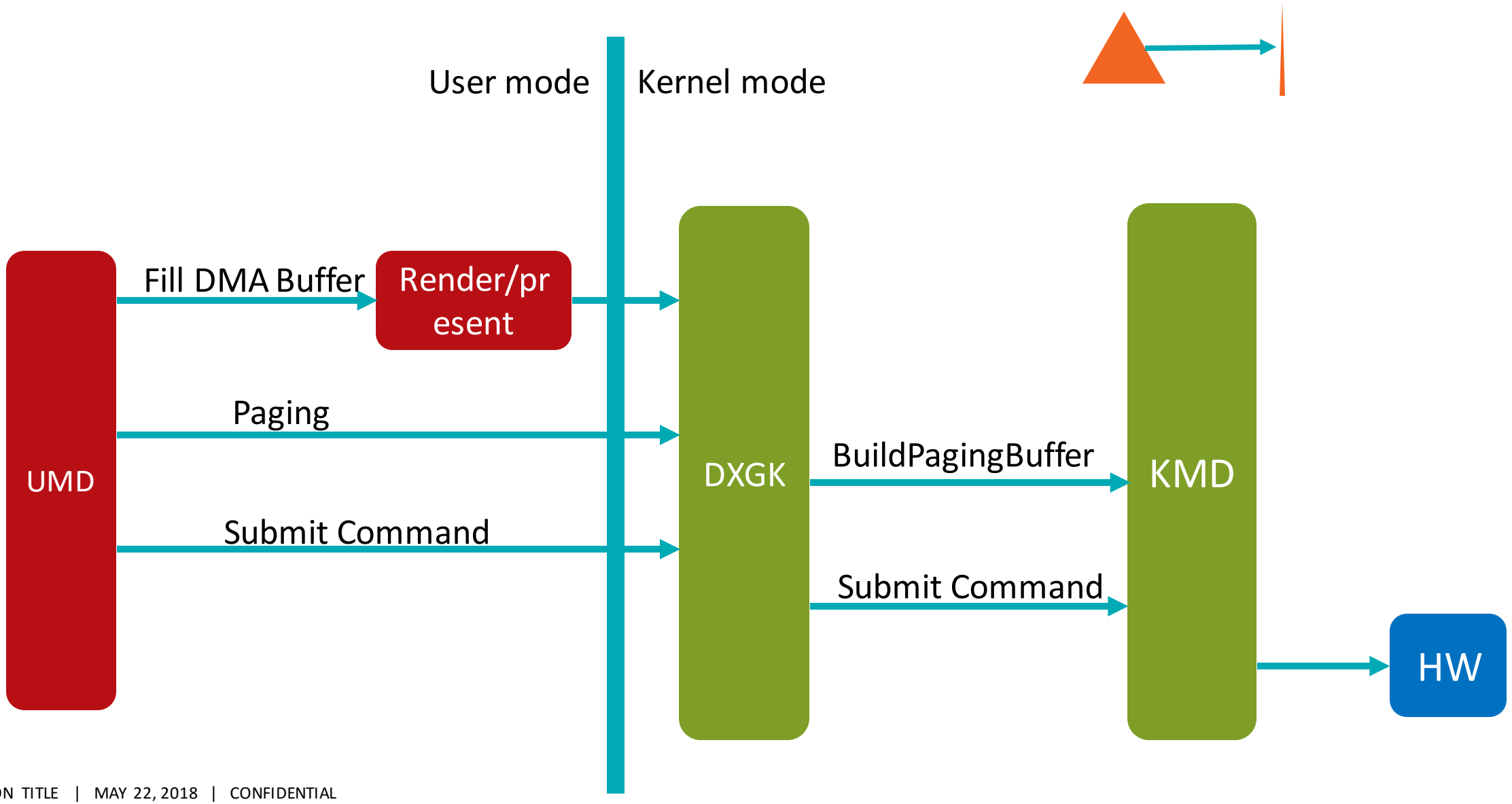
RENDER/PRESENT

PRIOR WDDM 2.0

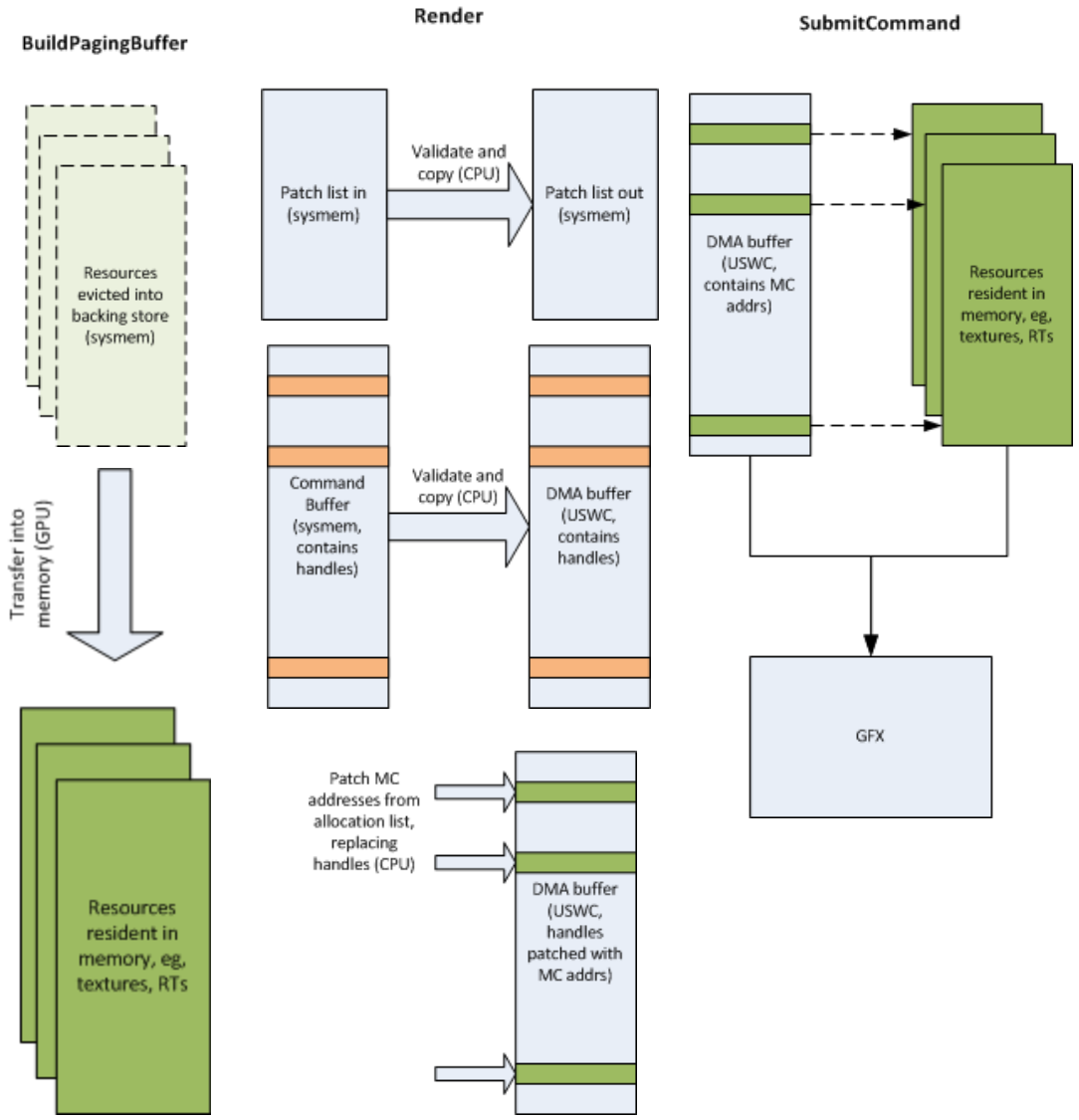


RENDER/PRESENT

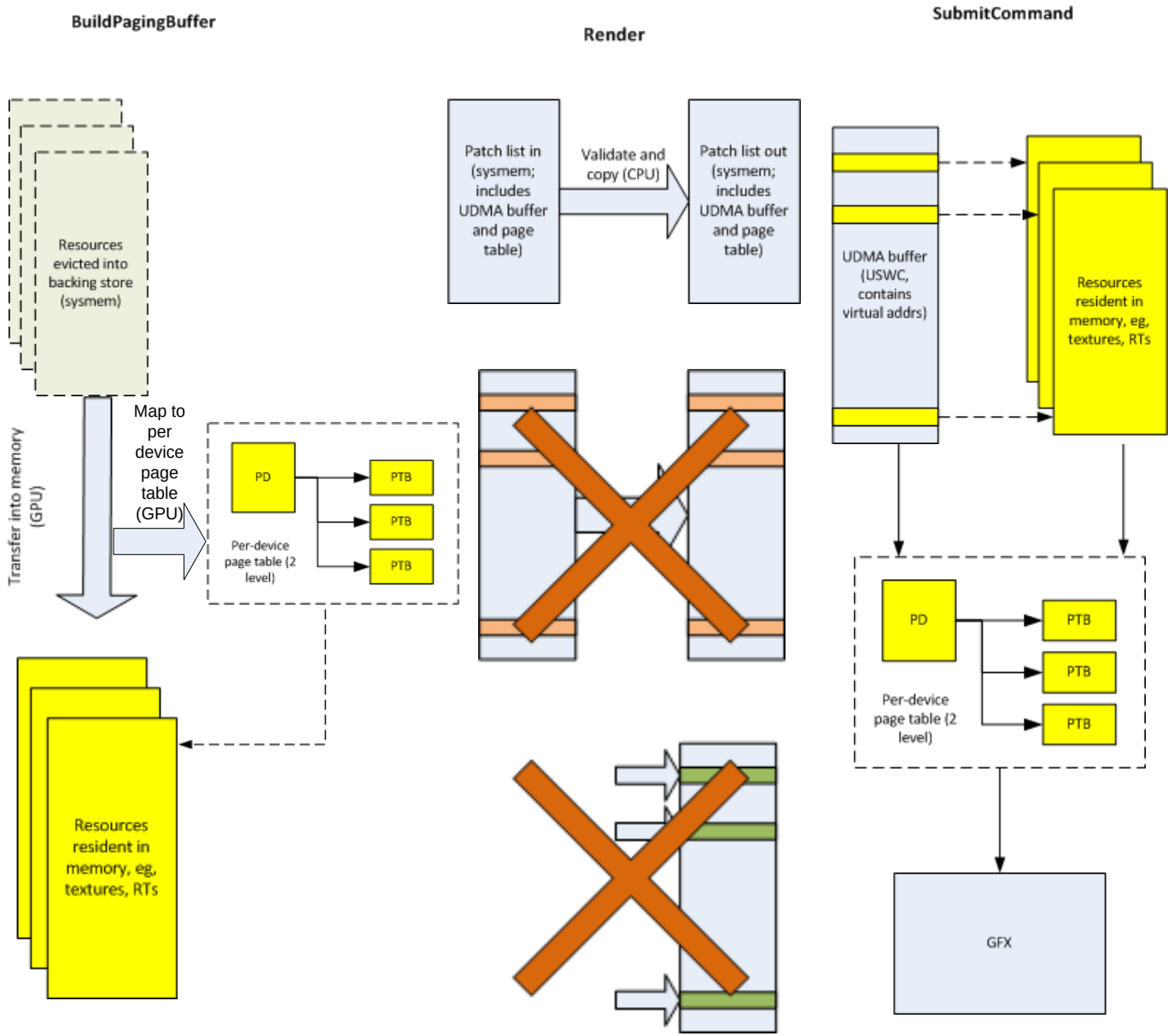
AFTER WDDM 2.0



BASIC MODEL



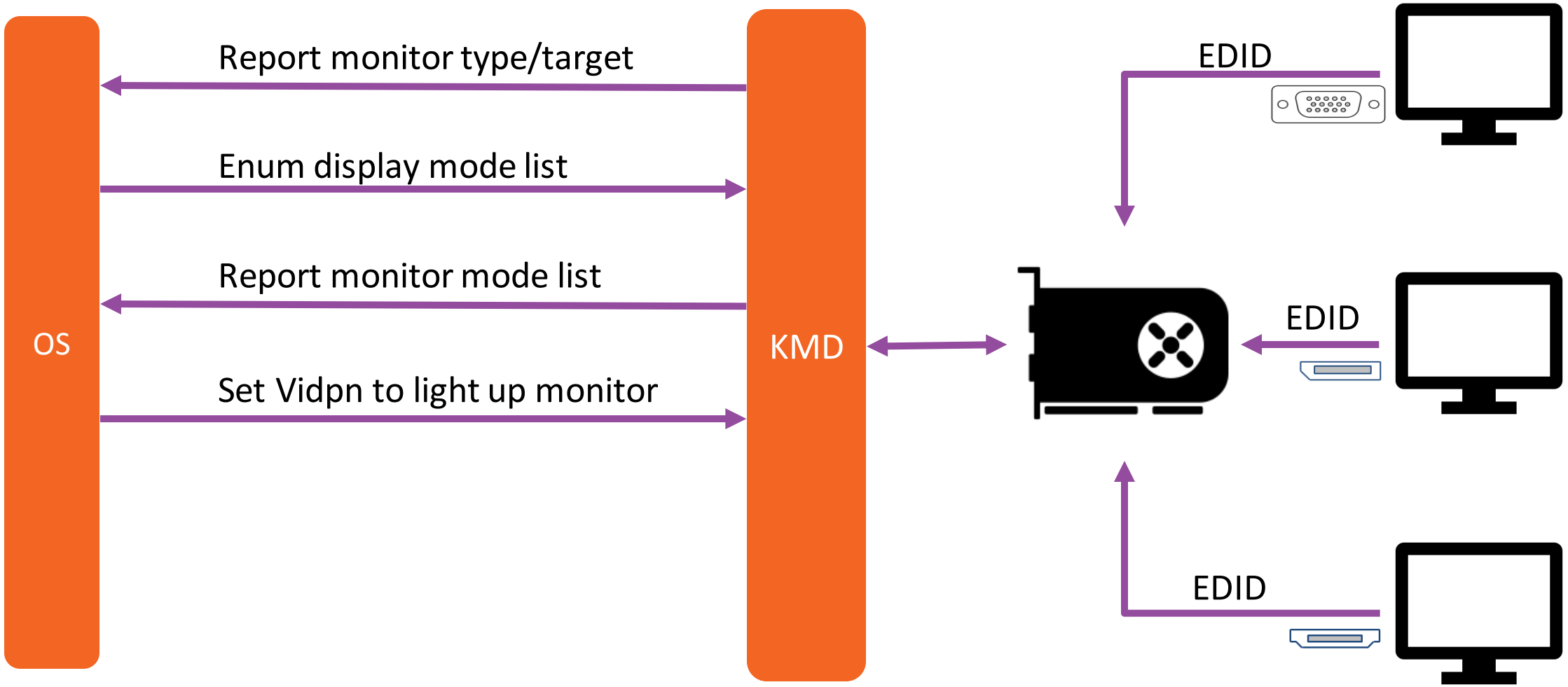
BASIC++ MODEL



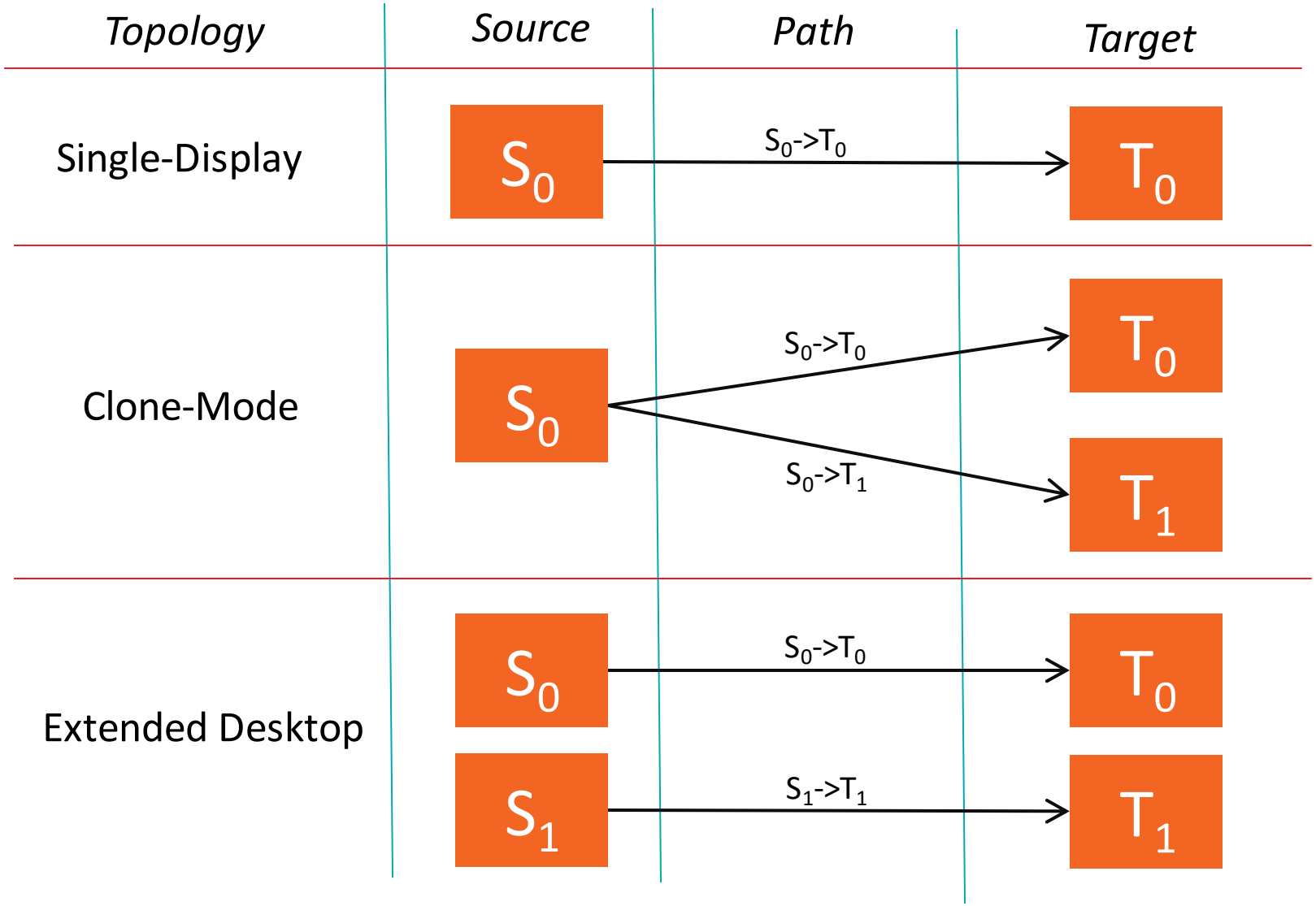
DISPLAY

DISPLAY

PRINCIPLE



VIDPN


Topology
Source
Path
Target
Single-Display
S_0
S_0->T_0
T_0
Clone-Mode
S_0
S_0->T_0
S_0->T_1
T_0
T_1
Extended Desktop
S_0
S_1
S_0->T_0
S_1->T_1
T_0
T_1

DISPLAY-RELATED DDIS AND VIDPNS

AMD

- Queries:
  - **QueryChildRelations:** Provides the OS with the type and capabilities of child devices
  - **QueryChildStatus:** Generally used by OS to ask driver about the connectivity of a particular child device (i.e., is there a monitor plugged in to this port?)
  - **QueryDeviceDescriptor:** Provide the OS with the EDID of a particular child device
  - **RecommendMonitorModes:** Provide the OS with a set of recommended modes for a monitor
- Mode enumeration:
  - **IsSupportedVidpn**: Given a VidPn, ask the driver if it's supported by the hardware
  - **EnumVidPnCofuncModality:** Given a set of VidPn constraints, ask the driver to enumerate all possible values for unpinned variables (e.g., if monitors A and B are in clone mode and A is 1920x1080, what are all the possible resolutions of B?)
- ModeSet, etc:
  - **CommitVidpn:** Program monitors according to the given VidPn (aka modeset, setmode, commit, mode switch)
  - **SetVidPnSourceVisibility:** Blank or unblank a display (i.e., turn it off or on)
  - **SetVidPnSourceAddress:** Start showing a new primary surface on the display (aka flip)

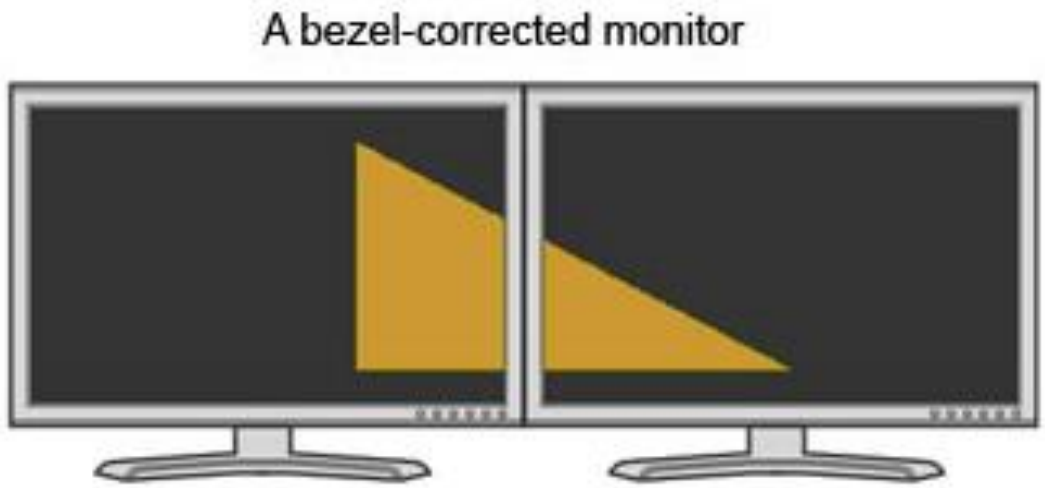
MGPU

BASE DRIVER "SPECIAL" FEATURES – MGPU

AMD

- **Proxy** is the primary base driver component responsible for multi-GPU features
- 2 MGPU configurations
  - **Multi-Adapter Mode:** At the KMD level, minimal collaboration between GPUs. The OS manages cross-GPU rendering logic, if any. Proxy is mostly operating in bypass mode. Examples:
    - IGPU+DGPU with one GPU idle
    - A+I Hybrid Graphics
      - Win10 supports display from either GPU – Win8 supports iGPU display only.
      - Includes support for detachable graphics
    - FLGL display solutions (large windowed application rendering on one GPU, copied by OS to other GPUs by OS for display)
  - **Linked Display Adapter (LDA) Mode:** Kernel mode driver asks the OS to "link" 2 or more adapters as a "chain" comprised of a Master adapter and one or more Slaves. The OS interacts only with the Master and the base driver internally manages the chain as desired. Proxy must dispatch DDI requests to the appropriate GPU(s) KMD(s)
    - **PowerXpress** (aka Switchable Graphics): A+A and A+I (Win7 only). dGPU only powers up when needed for high-performance rendering. "Local Display" allows displays to be driven from a slave GPU, which is otherwise impossible
    - **CrossFire:** Alternating Frame Rendering (AFR) where the UMD dispatches rendering requests to alternating GPUs to distribute the workload. Proxy inspects the GPU ordinal sent by the UMD to dispatch the render call to the appropriate KMD.
    - **MGPU SLS:** Single GPU renders, multiple GPUs display portions of an SLS surface

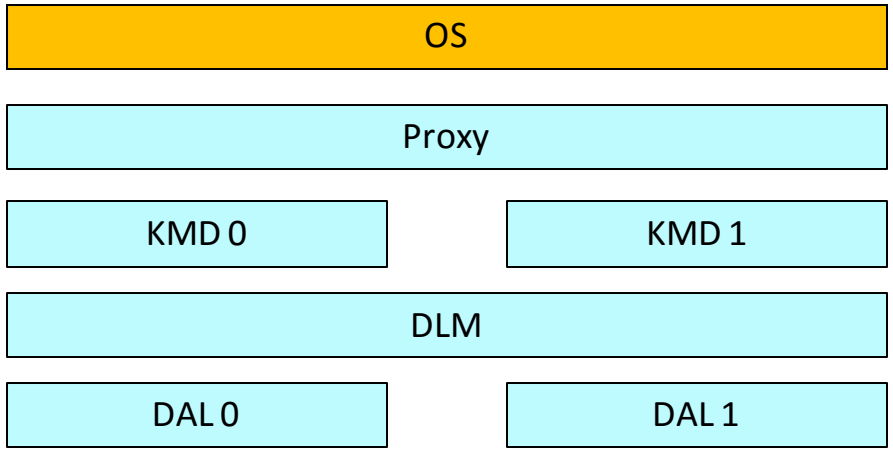
EYEFINITY



BASE DRIVER MULTI-GPU ARCHITECTURE

AMD

- Proxy and DLM are global objects
- Other components are created per-GPU
- OS thinks Proxy is the KMD driver
- KMD Driver thinks Proxy is the OS
- Most MGPU logic (PowerXpress, CrossFire) is in Proxy
- Some MGPU logic in DLM/KMD
- Private communication channels are available between KMD(s) and Proxy



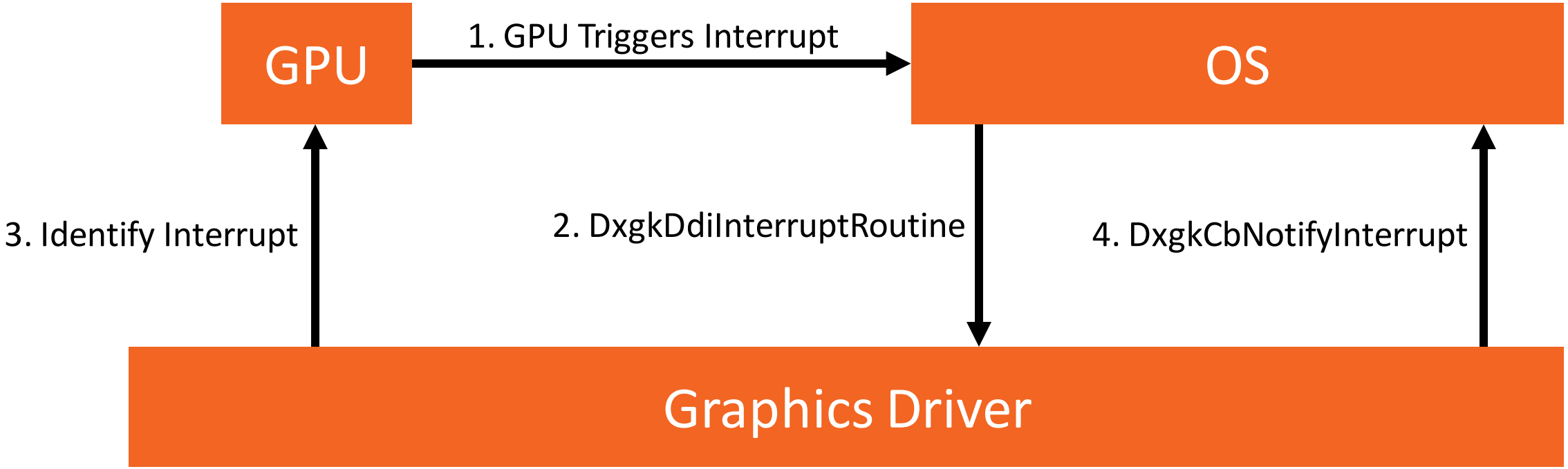
INTERRUPT

INTERRUPTS

AMD

- Asynchronous events triggered by hardware to notify listeners to take some action
- Graphics hardware uses many, but of main importance are:
  - **VSYNC Interrupts:** Issued on every VBLANK (e.g., every 16.6ms on a 60Hz monitor)
  - **DMA Complete Interrupts:** Issued after the GPU has completed execution of a DMA buffer
  - **DMA Preempted Interrupts:** Issued after the GPU has preempted a DMA buffer
  - **Page Fault Interrupts:** Issued upon invalid access to a video memory location
- Interrupts are reported to the KMD via the OS (not directly from hardware)
  - OS calls DxgkDdiInterruptRoutine:
    - If the KMD does not own the interrupt in question, return FALSE
    - If the KMD does own the interrupt in question:
      - Dispatcher forwards to IRQ MGR
      - IRQ MGR will identify the type/source of the interrupt and acknowledge it (de-assert the interrupt)
      - IRQ MGR will call back into the appropriate driver subcomponents to handle the interrupt
      - The drive component (e.g., Interrupt Manager) will either handle/respond to the interrupt within the ISR or call a DPC for further processing at a lower IRQL
      - Typically InterruptManager will call DxgkCbNotifyInterrupt to report the interrupt back to the OS (e.g., VSYNC, DMA)

INTERRUPTS

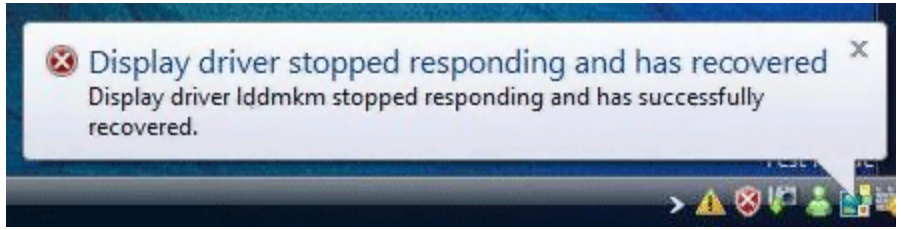

GPU
1. GPU Triggers Interrupt
OS
3. Identify Interrupt
2. DxgkDdiInterruptRoutine
4. DxgkCbNotifyInterrupt
Graphics Driver

TDR

TDR

AMD

- Timeout Detection and Recovery happens when the KMD does not report an expected interrupt to the OS within a pre-specified amount of time (e.g., 2 seconds for DMA complete interrupts). Examples:
  - After a call to SetVidPnSourceAddress (flip), the KMD must report a “CRTC VSYNC” interrupt within the timeout interval (function of refresh rate). That interrupt must:
    - Contain a primary surface address matching the one passed in to SetVidPnSourceAddress
    - Originate from the preferred VidPn Target associated with the VidPn Source to which SetVidPnSourceAddress was directed
  - After a call to SubmitCommand(Virtual), the KMD must report a “DMA Completed” interrupt within the timeout interval (2 seconds). That interrupt must:
    - Contain the submission fence ID of the DMA buffer submitted
    - Contain the node/engine ordinals of the node/engine to which the DMA buffer was submitted
- If an expected interrupt is not reported in time, TDR is initiated
  - Detection: DxgkDdiQueryCurrentFence
  - Reset: DxgkResetFromTimeout/RestartFromTimeOut
  - On Win8+ the OS will attempt a per-engine reset before resetting the entire ASIC
  - Too many TDRs = BSOD (usually points to Proxy driver)
  - May cause application crash, but avoid hard hang



FLIP/PRESENT

FLIP/PRESENT

AMD

- A primary surface is an area in memory for storing a final composited image to be displayed on screen.
- Typically (but not always) 2 primary surfaces in play (per desktop):
  - **Front Buffer:** On-screen surface, not to be rendered to
  - **Back Buffer:** Off-screen surface used to composite a frame to be displayed
- A collection of front and back buffers is called a **Swap Chain**
- **Flipping** is the process of changing the address of the primary surface that is currently on screen (i.e., swapping the addresses of the front buffer and back buffer)
- When UMD calls PresentCb to say a frame is ready, KMD sees the following:
  - **DxgkDDIPresent (optional):** Copy contents of a source surface to a primary destination surface (usually). OS provides the src and dst surface handles, "instructions" on how the copy should be performed, and an empty DMA buffer. KMD fills the DMA buffer with the appropriate GPU commands to execute the copy as requested. The OS will then submit the buffer to the GPU. Typically, the destination is an off-screen primary surface back buffer.
  - **DxgkDDISetVidPnSourceAddress:** This is the real "flip" for the KMD, when the memory address that the display controller is currently reading from changes. The OS tells KMD which desktop is being flipped (i.e., which VidPnSource), the new surface address, and flags to specify the type of flip (e.g., immediate vs VSYNC)

TYPES OF FLIPS

AMD

VSYNC Flip

- "Vertical Synchronization" flip, where the screen content is refreshed at the vertical blanking interval
- Ensures a "tear-free" experience
- SetVidPnSourceAddress immediately returns to the OS, even though the flip hasn't happened.
- The provided address becomes the "pending address" and when the next VSYNC occurs (meaning the display has just finished scanning a full frame), the flip actually happens.
- The OS expects a VSYNC interrupt with the new address from the appropriate monitor soon after a SetVidPnSourceAddress call.

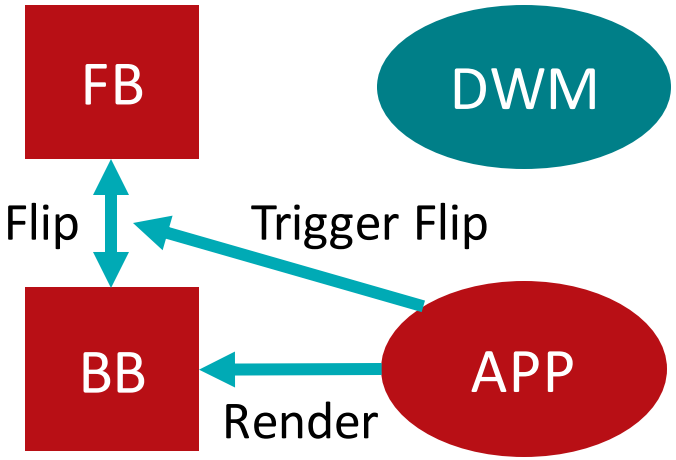
Immediate/HSYNC Flip

- "Horizontal Synchronization" flip, where the screen content can be refreshed at the end of any scan line
- Tearing is often visible when the flip happens while the scan line is mid-frame
- Only (?) used in exclusive full-screen mode and for secondary displays in a clone-mode topology
- SetVidPnSourceAddress doesn't return until the flip actually happens. If the DCE scanline is currently mid-screen, tearing may be observed as the display controller begins reading from the new location

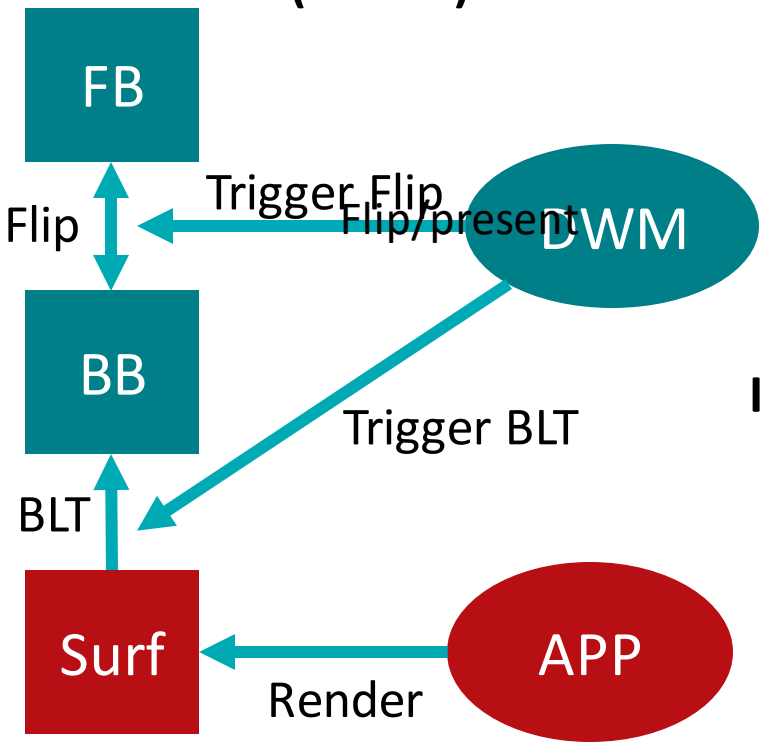
TYPES OF FLIPS

AMD

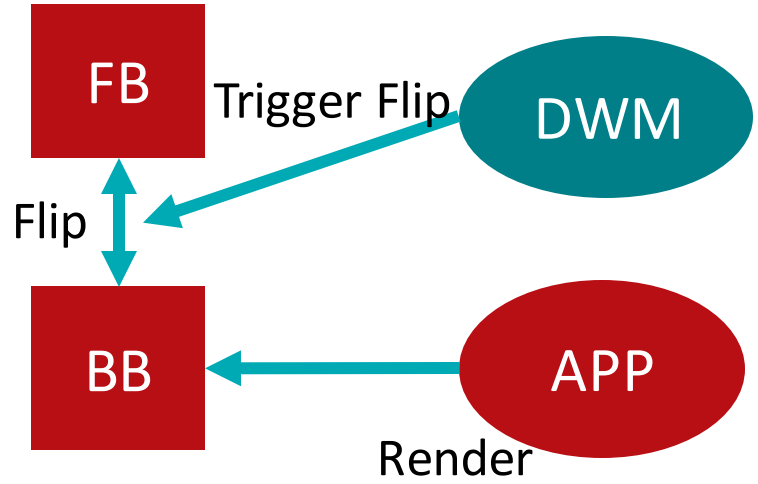
Full-screen Exclusive Mode (VSYNC or HSYNC)



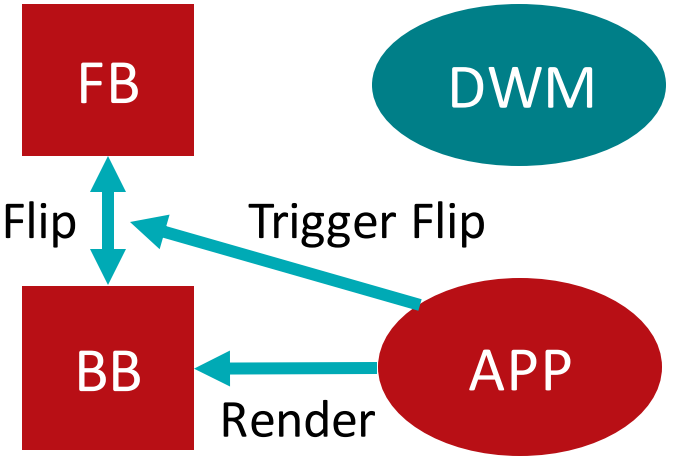
"Classic" Windowed-Mode (VSYNC)



DirectFlip Windowed-Mode(VSYNC)



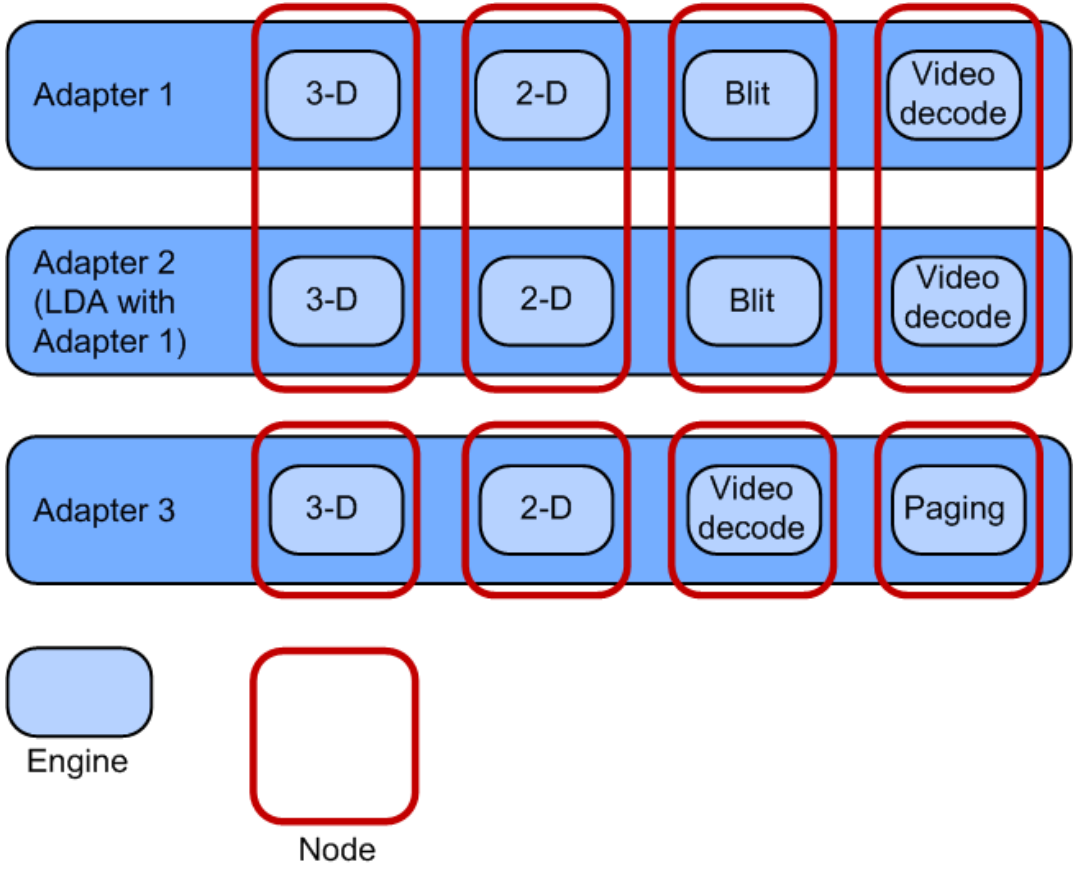
IndependentFlip Windowed-Mode (VSYNC)



GPU ENGINE

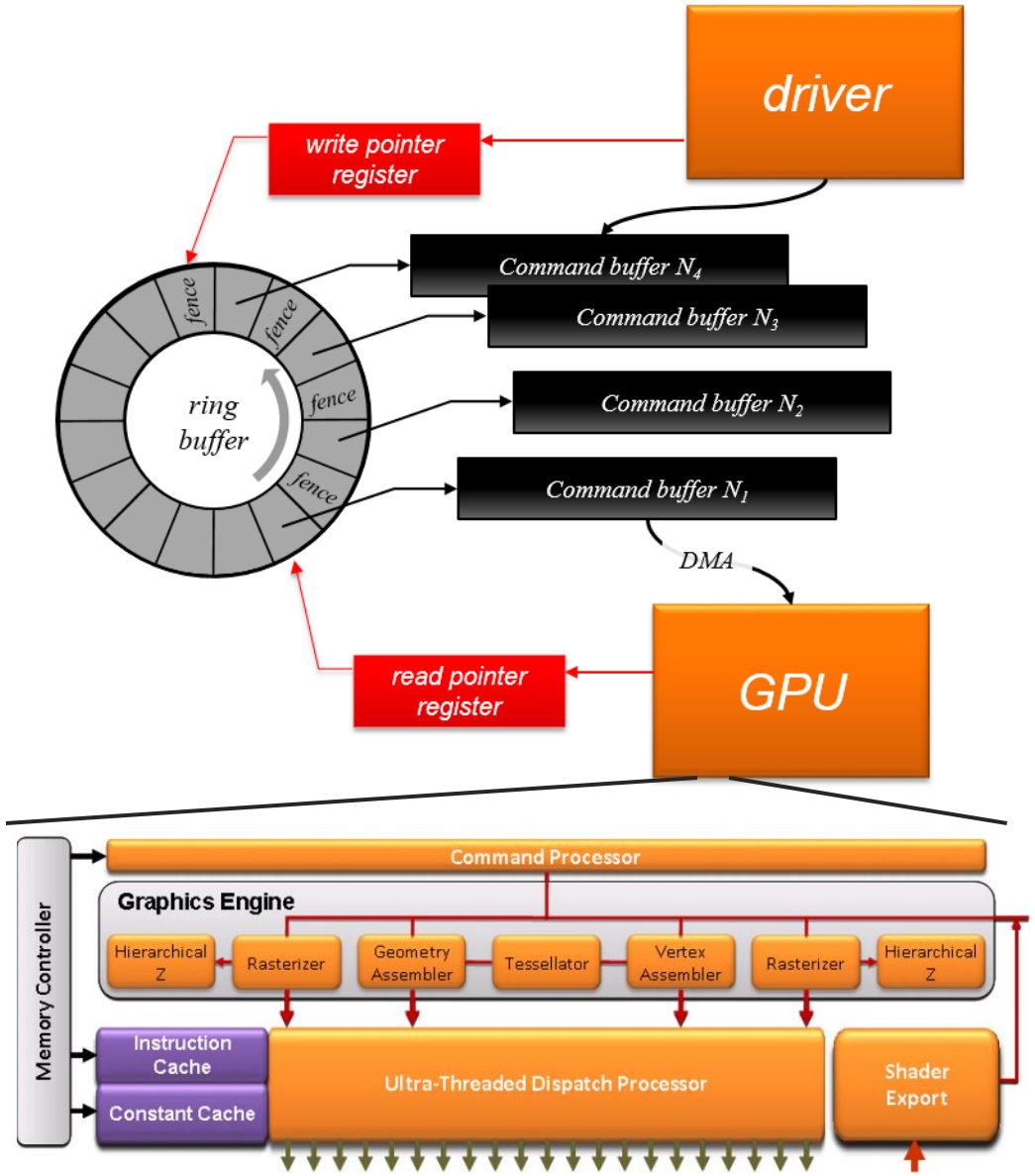
ENGINES, NODES, CONTEXTS, ETC.

- A **Node** is a software representation of one or more engines
  - *Typically* a 1:1 mapping, except for LDA chains
  - Nodes/Engines enumerated by our adapter:
    - **3D:** Graphics
    - **SDMA:** 2 engines for paging/copying (e.g., SW CF)
    - **Compute:** Usually for OCL
    - **UVD:** Video Decode
    - **VCE:** Video Compression Engine (Encoding)
    - **PSP:** Platform Security
- A **device** creates a **context** against a particular node and work submitted to that context will be executed by the engine(s) associated with the node
- One node/engine can have multiple contexts created on it and the OS handles scheduling between contexts
- Engines have a **ring buffer** where submissions are queued and fetched by the engine for execution…

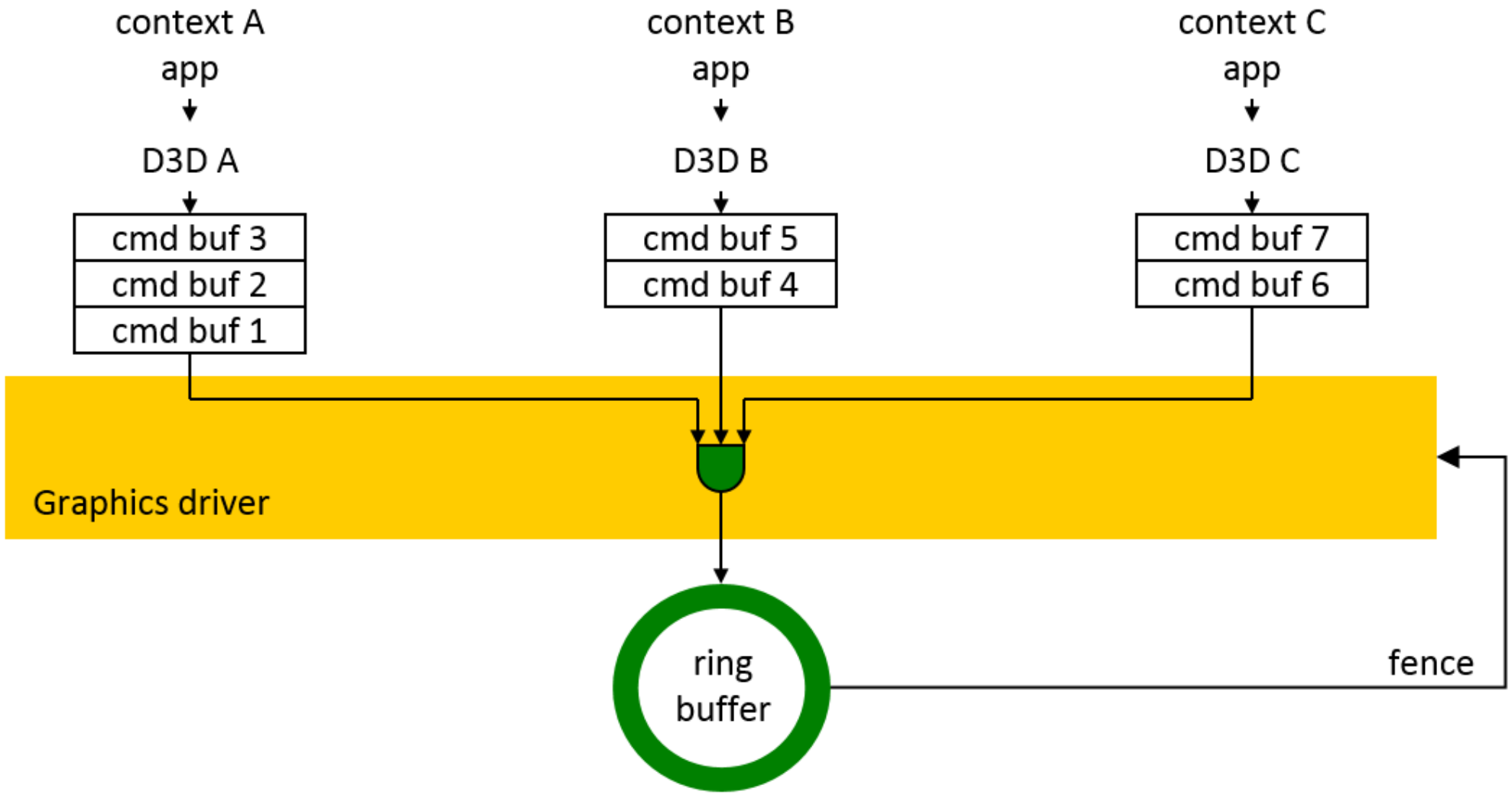


RING BUFFERS AND THE CP

- Ring Buffers are the interface between SW and HW, where the driver's job ends and the GPU's job begins
- Allocated in linear GPU-accessible system memory.
- Once the end is reached, it "wraps around"
- The driver (host/CPU) writes packets into the ring buffer at the location of the write pointer
- The engine (GPU) reads packets from the ring buffer at the location of the read pointer using its **Command Processor (CP)**
- When the engine is idle, the read pointer and the write pointer should be the same
- Each packet has a unique fence associated with it, which the GPU can use to indicate completion to the driver via an interrupt
- A typical entry in the ring buffer is a pointer to an **Indirect Buffer**, which is a DMA buffer containing PM4, which the engine fetches and executes



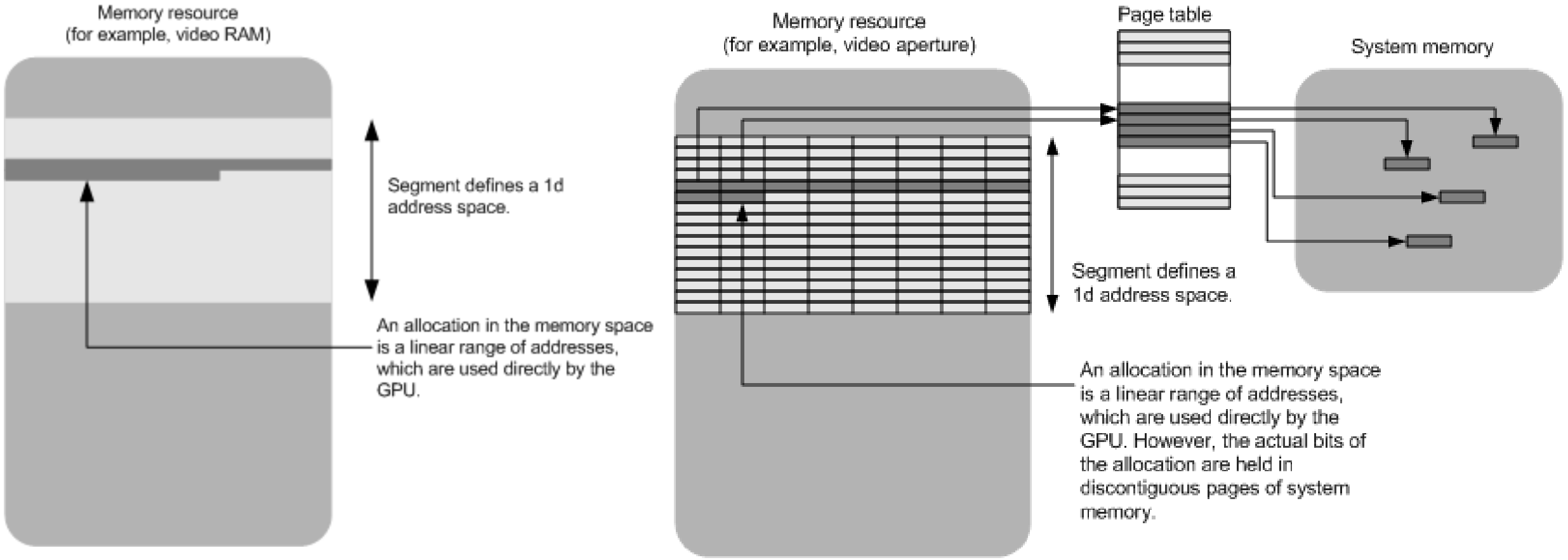
MULTIPLE CONTEXTS, ONE ENGINE



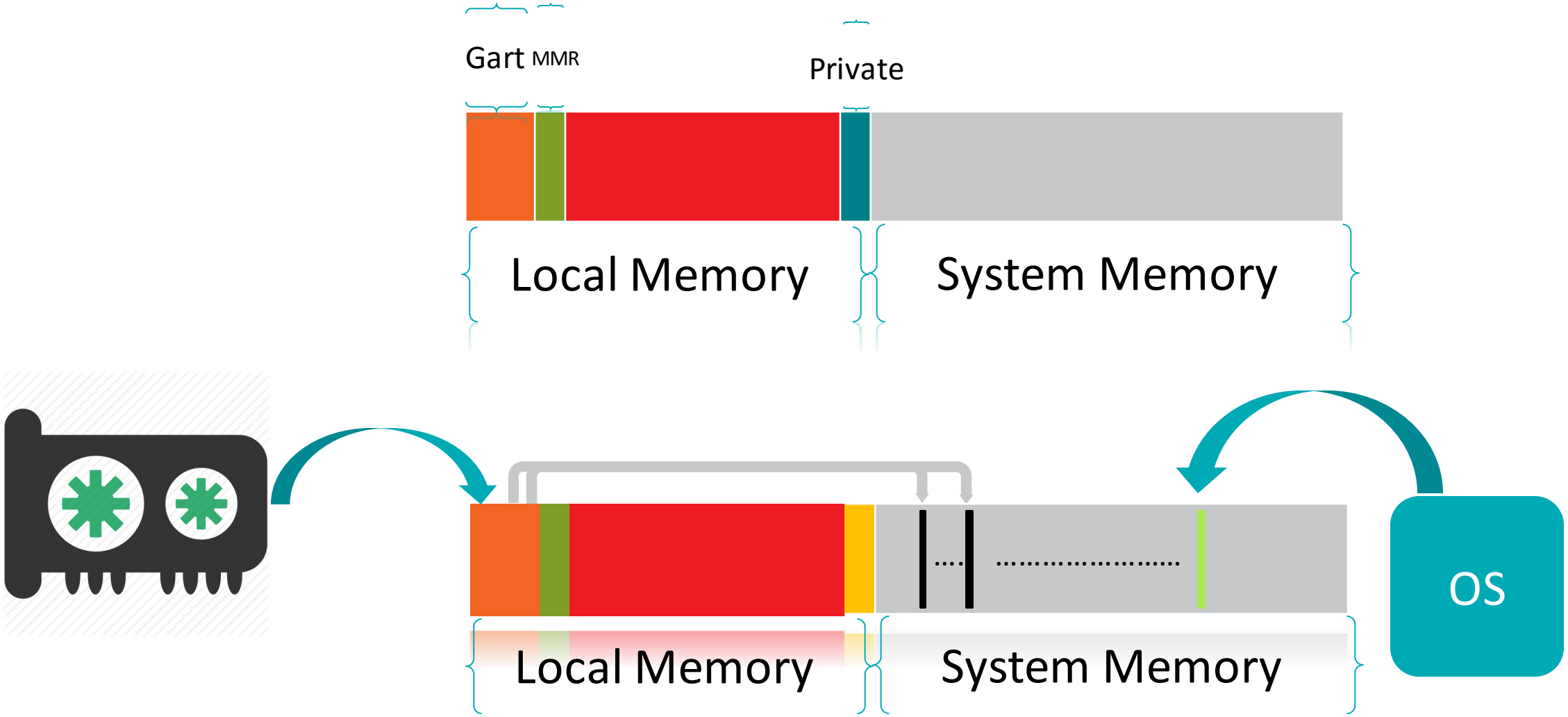
MEMORY IN THE KMD

MEMORY IN THE KMD

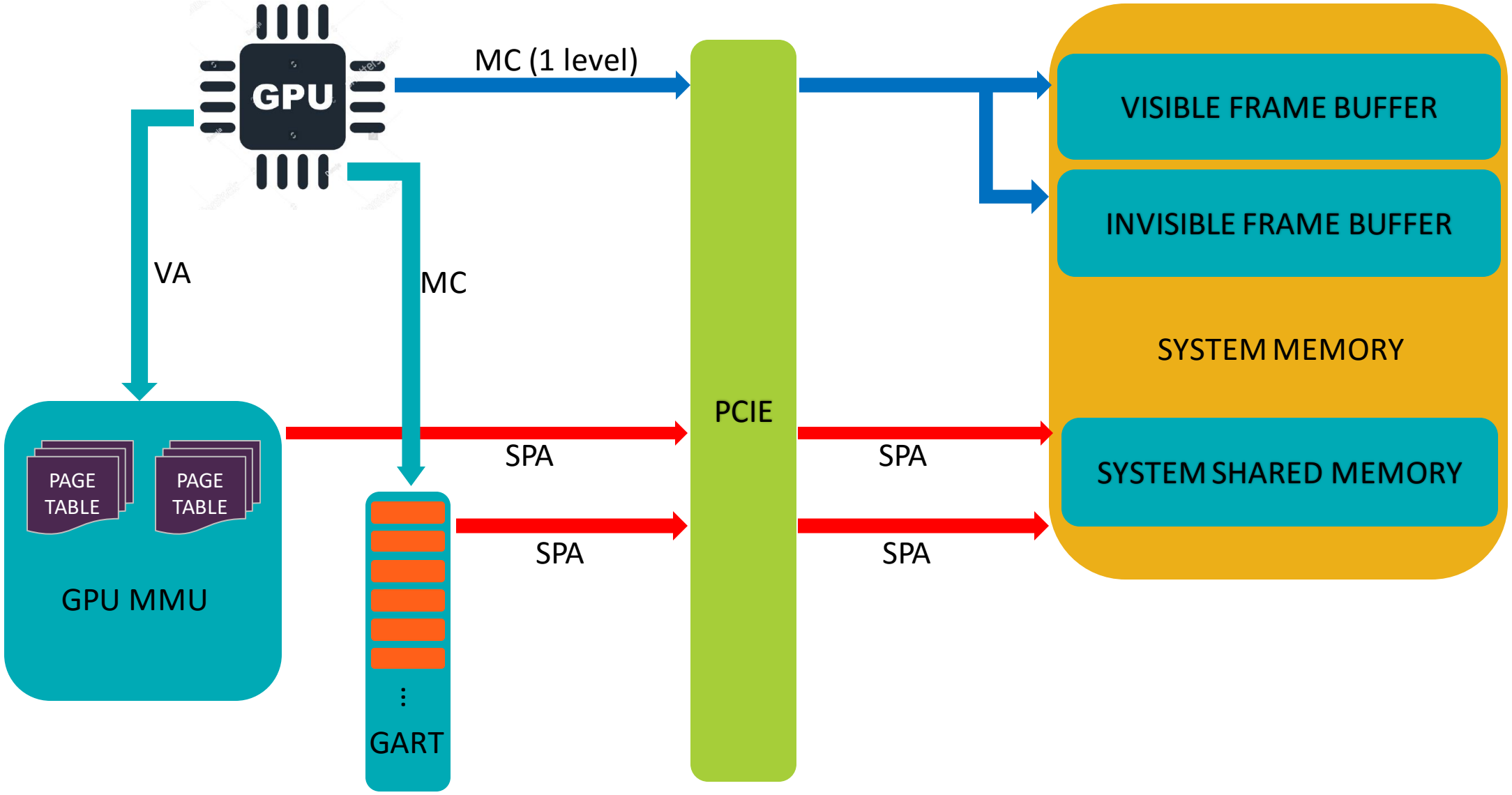
GPU-ACCESSIBLE MEMORY – SEGMENTS (WDDM1.X)



MEMORY SEGMENT



BEFORE IOMMU



MEMORY IN THE KMD

ALLOCATIONS

- **“Allocation”** is often synonymous with “**buffer**” or “**surface**”
- When an allocation is “created” by an app, that action in itself does not use any GPU memory. Creating an allocation just defines properties of an allocation. For example:
  - Handle (hAllocation) – a kind of virtual address for the allocation
  - Size/Dimensions (pitch, aligned height, rotation, etc.)
  - Pixel Format/Tiling Mode (Linear, 1D, 2D, thin/thick, etc.) describing the data layout of a surface
  - Supported/Preferred/Eviction Segments (read and write)
  - Priority (in case of segment contention)
  - Usage Hints (what will the allocation be used for? Vertex Buffer? Index Buffer? Texture?)
  - Flags
    - **IsPrimary:** For primary surfaces. Ensures a non-pageable buffer handle, displayable by DCE
    - **CPUVisible:** Required for lockable allocations
    - **PermanentSystemMem:** Always keep a copy of the allocation in system memory, even when in a FB segment
    - **Cached:** Should the backing store for the allocation be in cached memory?
    - **LinkedMirrored:** Allocations replicated across all adapters in LDA chain. Paging happens on all adapters (we always set this for LDA)
    - **AccessedPhysically:** Used in WDDM 2.0 to indicate the allocation must be allocated contiguously in FB memory/GART

MEMORY IN THE KMD

PAGING AND RESIDENCY

AMD

- **The OS allows more video memory resources to be created than there is physical video memory available**
  - Made possible by providing an independent virtual memory space to every application and managing the physical segments by paging allocations in/out of memory as necessary
  - At a given time, only a subset of all created video memory resources will actually be resident in memory

- OS ultimately controls residency, but driver influences policy with allocation hints, segment preferences, etc.

- OS uses the GPU to move allocations around, fill them with patterns, etc., but doesn't know the GPU-specific instructions (PM4, in our case) used to perform these actions. Therefore, to trigger a paging operation:
  - OS calls KMD **DxgkDdiBuildPagingBuffer** with a request (e.g., move 2MB of memory from location X to location Y)
  - KMD fills in the provided DMA buffer (aka the "Paging Buffer") with the GPU instructions that will execute the operation requested and returns to the OS
  - OS calls back to KMD's DxgkDdiSubmitCommand DDI with the populated DMA buffer
  - KMD submits the DMA buffer to the SDMA engine to perform the paging operation

MEMORY IN THE KMD

BUILD PAGING BUFFER

AMD

- BuildPagingBuffer has several options. The fundamental ones are:
  - **Fill(Virtual):** Initial setup of an allocation in local memory (fill region with a specified pattern)
  - **Transfer(Virtual):** Transfer contents of memory location to another location (usually to/from non-local)
  - **Discard:** Evict an allocation from local memory (without transfer to a backing store)
  - **Map/Unmap:** Maps/Unmaps pages of system memory (using an MDL) into/out of the aperture segment

- Physical system memory pages must be mapped into the GART aperture using the BPB Map operation before an allocation can be transferred to non-local memory

- In WDDM2.0, the OS also manages page tables and directories that reside in GPU segments, but are stored in a HW-specific format. Therefore, BPB is also used to update/manage those page tables:
  - **UpdatePageTable:** Modify one or more entries of a page table/directory (using CPU or GPU)
  - **FlushTlb:** Flush the translation lookaside buffer (important after a PT has been updated)
  - **NotifyResidency:** Notification that residency has changed (AccessedPhysically allocations only) after fill/transfer

MEMORY IN THE KMD

WDDM1.3 VS 2.0 RESIDENCY

AMD

- **How does the OS know which allocations must be resident at a given time?**
  - **WDDM 1.3:**
    - **OS ensures allocations are resident when they are about to be used in a command/DMA buffer**
    - Every submission from the UMD is accompanied by an **allocation list** with one entry per allocation referenced in the command buffer
    - The OS reviews the allocation list and pages in any non-resident allocations (using a series of BPB calls) before submitting to the GPU
    - Paging happens in the Scheduler thread context (same thread as submissions to the GPU)
  - **WDDM 2.0:**
    - **UMD controls residency by calling MakeResident/Evict. UMD must ensure all allocations referenced in a Command/DMA buffer are requested to be resident before submission**
    - The OS reviews the residency list for a device and pages in any non-resident allocations (using a series of BPB calls) before scheduling any contexts belonging to that device for execution
    - If a UMD requests too much memory be made resident at once, the OS asks it to trim its residency list.
    - No allocation list required! (as long as the engine supports virtual addressing)
    - Paging happens in independent system paging process context

MEMORY IN THE KMD

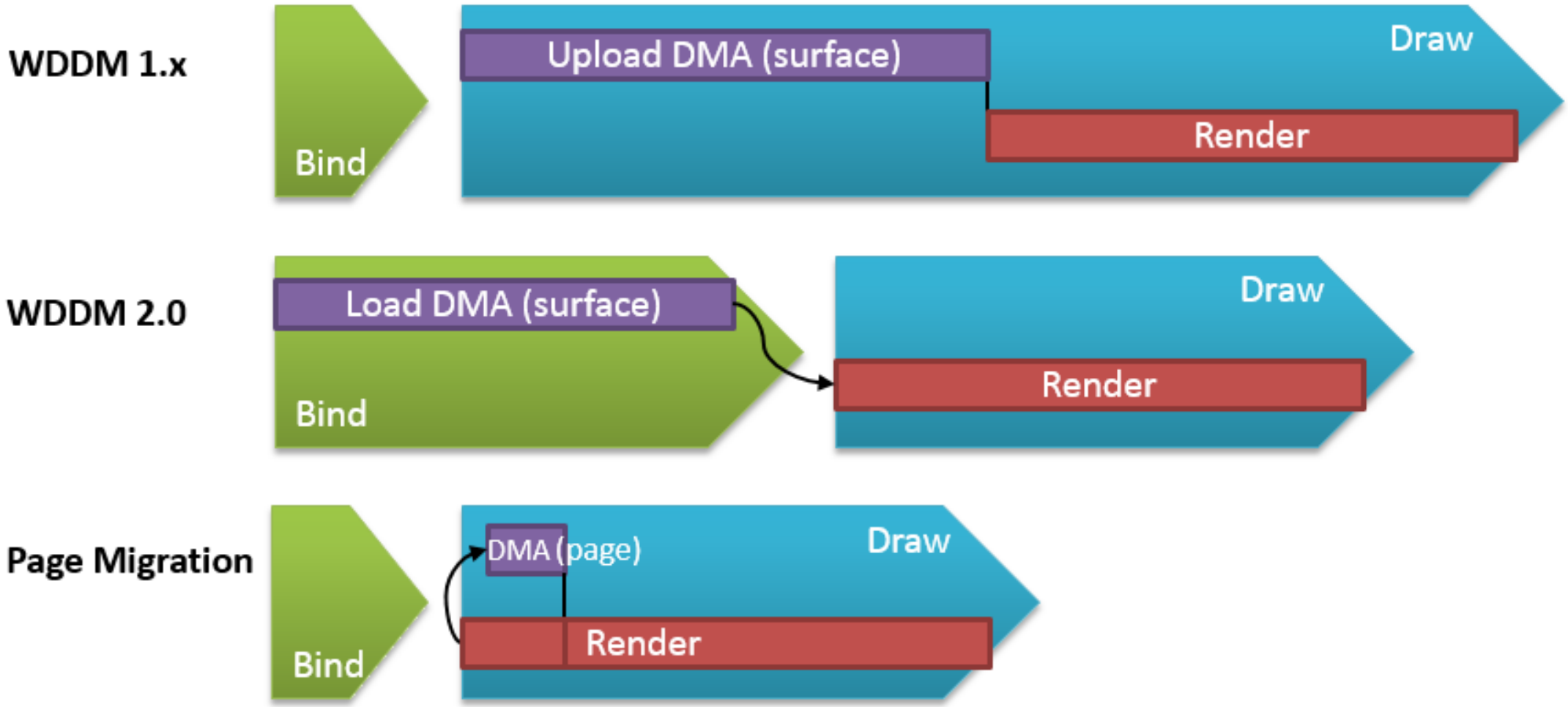
PAGE MIGRATION (FIJI AND GREENLAND)

AMD

- The OS does not always do a reliable job of managing residency between local (fast but small) and non-local (slow but large) memory
- Exacerbated by using small amounts of HBM (high-bandwidth memory)
- “Page Migration” is a KMD/HW feature that aims to take the OS out of the equation and optimally auto-migrate resources on demand
- Already partially implemented on Fiji
  - KMD will sometimes override OS residency decisions by evicting low-priority resources (victims) and privately paging in high-priority resources
  - Only enabled for very high resolutions
- Will be fully enabled on Vega10+
- Benefits include residency granularity at the page level and late-stage resource binding

MEMORY IN THE KMD

PAGE MIGRATION (FIJI AND GREENLAND)



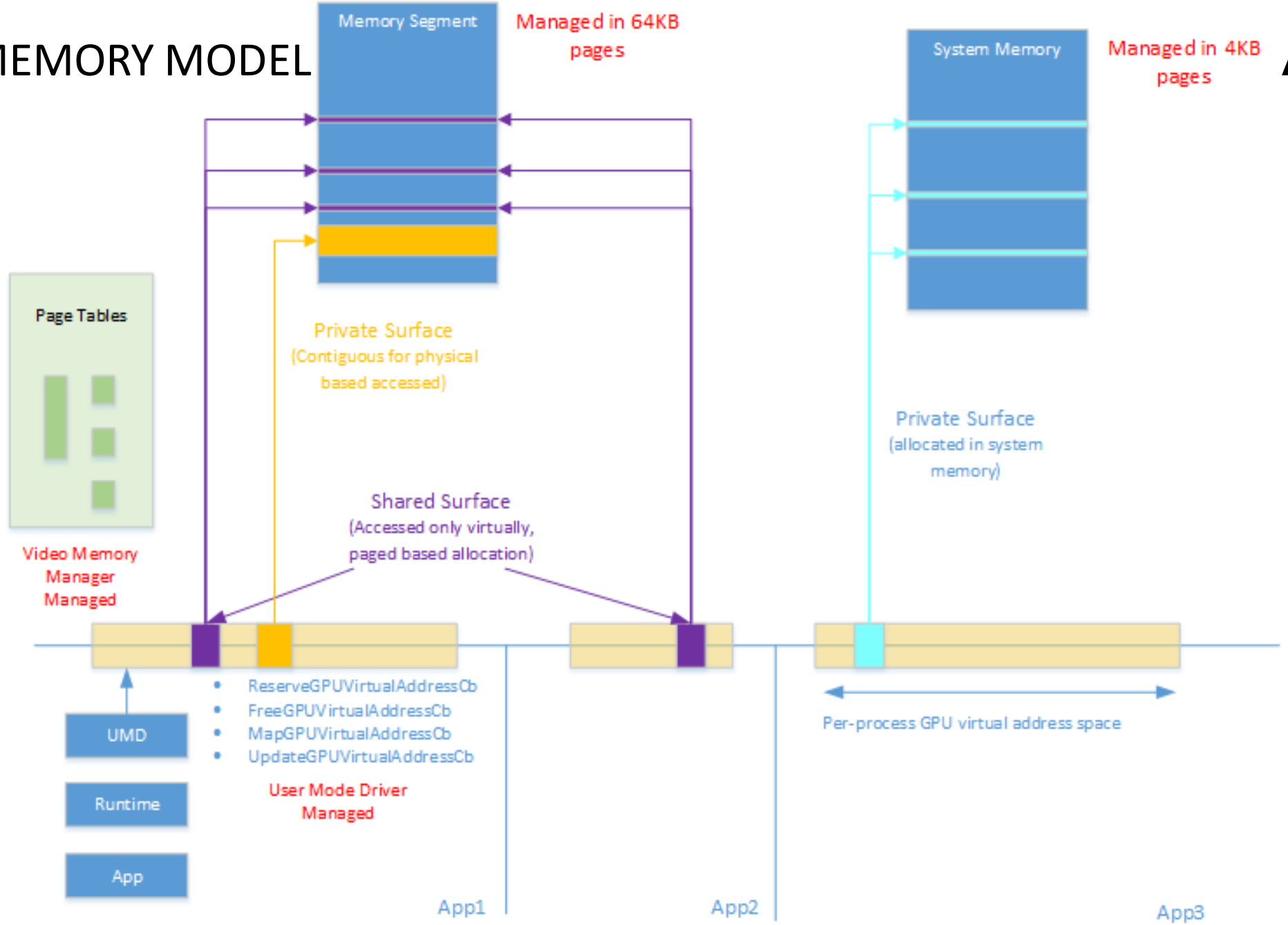
MEMORY IN THE KMD

PHYSICAL VS VIRTUAL SUBMISSION

AMD

- The PM4 instructions placed in command buffers by the UMD contain references to a variety of allocations (textures, vertex buffers, index buffers, render targets, the primary surface, etc.). Those references can be:
  - **Physical Addresses:** the GPU MC address corresponding to the location of the referenced allocation in local memory
  - **Virtual Addresses:** if the engine is capable of virtual submission, it can be programmed to use page tables (typically stored in local memory) to translate VAs into their corresponding physical addresses
- **Physical Submission (WDDM 1.x Basic model):**
  - UMDs use an allocation handle to refer to allocations in the command buffer
  - During submission, after KMD has copied the command buffer to a DMA buffer in the Render DDI, the OS calls **DxgkDdiPatch**, asking KMD to replace all virtual references with their physical addresses (provided in a "patch list")
  - The OS submits the patched DMA buffer to the GPU
- **Virtual Submission (WDDM 1.x Basic++ or WDDM 2.0 model)**
  - UMDs use unique virtual addresses to refer to allocations in the command buffer
  - During BPB calls, page tables on the GPU are updated to keep an accurate mapping between VAs and their physical pages
  - The DMA buffer is submitted to the GPU with the original VAs used to reference allocations. The VM hardware built into the GPU is able to dereference those VAs using the page tables, which are resident in memory

WDDM 2.0 MEMORY MODEL



Q & A

DISCLAIMER & ATTRIBUTION

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION