# INTRODUCTION TO DPDK

Network Platforms Group – September 2015

# Legal Disclaimer

**General Disclaimer:**

© Copyright 2015 Intel Corporation. All rights reserved. Intel, the Intel logo, Intel Inside, the Intel Inside logo, Intel. Experience What's Inside are trademarks of Intel. Corporation in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others.

**Technology Disclaimer:**

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at [intel.com].

**Performance Disclaimers (include only the relevant ones):**

Cost reduction scenarios described are intended as examples of how a given Intel- based product, in the specified circumstances and configurations, may affect future costs and provide cost savings. Circumstances will vary. Intel does not guarantee any costs or cost reduction.

Results have been estimated or simulated using internal Intel analysis or architecture simulation or modeling, and provided to you for informational purposes. Any differences in your system hardware, software or configuration may affect your actual performance.

# Topics

Overview of DPDK

- Why DPDK – PMD vs Linux interrupt driver, memory config, user space.
- Licensing
- Packet processing concepts
- DPDK component libraries
- Memory IA – NUMA, Caching, huge pages, TLBs on IA
- Memory DPDK – mem pools, buffers, allocation etc.

# Data Plane Development Kit (DPDK)

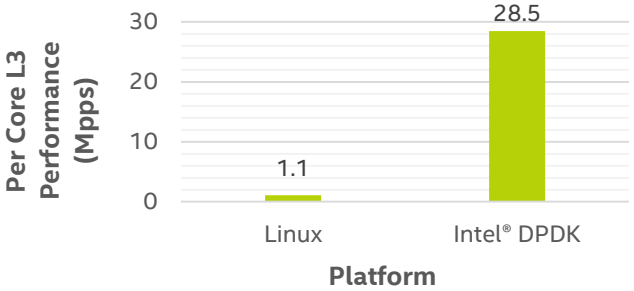
- **Big Idea**

  *Software solution for accelerating Packet Processing workloads on IA.*

  - Delivers 25X performance jump over Linux
  - Comprehensive Virtualization support
  - Free, Open Source, BSD License
  - Enjoys vibrant community support

- **Deployment Models**

- **Performance**




- **Commercial Support**

WIND RIVER

Disclaimer: Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

# Where is DPDK today?

Free, Open-sourced, community driven, BSD Licensed
- Git: http://dpdk.org/git/dpdk

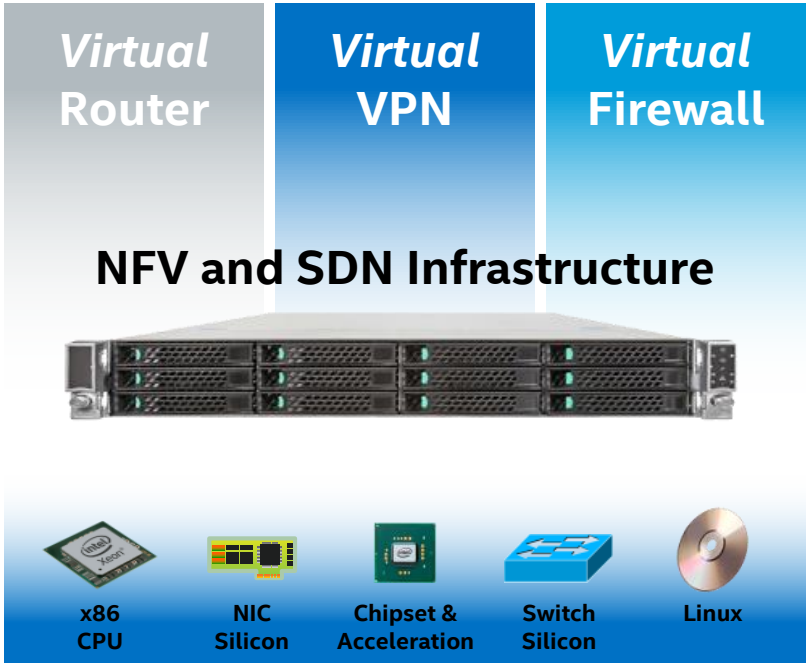
Multiple CPU architectures supported (on dpdk.org)
- Intel x86_64, ia32, Power 7/8, Tilera (EZChip)

Multiple vendor NICs supported in open source
- R2.1: Intel, Cisco (VIC), Mellanox, Broadcom (Qlogic), Chelsio
- R2.2: +NetFPGA, +others

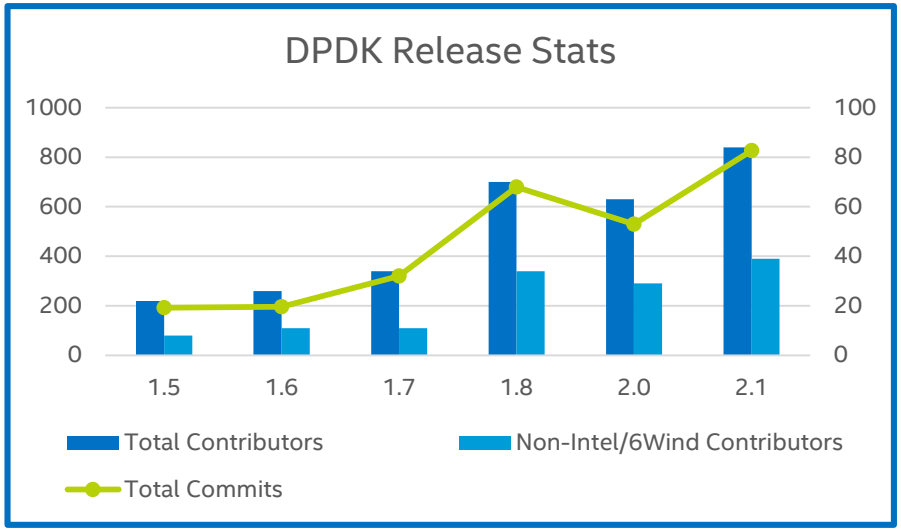
Multiple OS Distributions

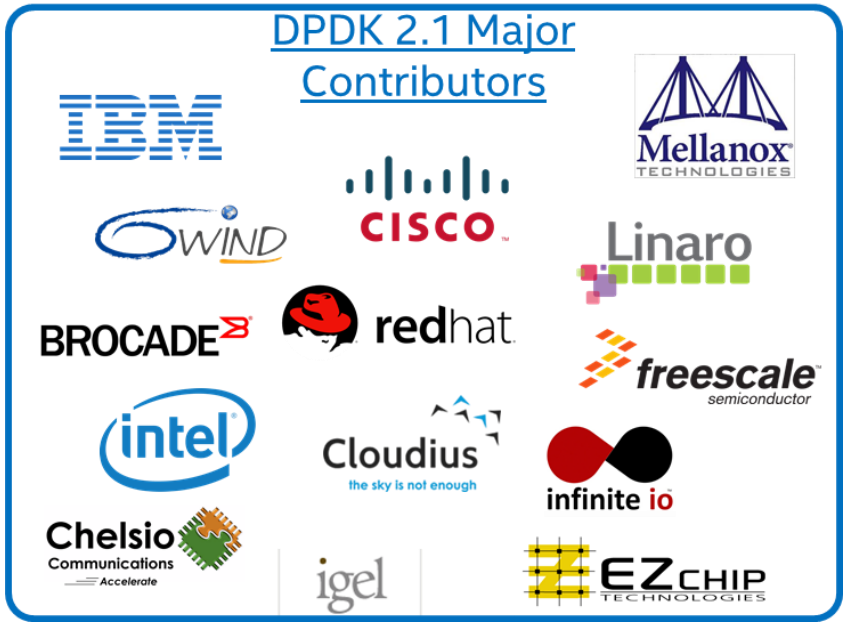
Multiple virtualized environments
- KVM, VMware, Xen

Some closed source drivers/ports



# DPDK Community




## dpdk.org

- Multiple maintainers and patch policy established
- Lack of community support for TSC or moving to a Linux Foundation project

## 1.53M lines of code - 128 contributors

- Test framework and test cases expediting quality of patches and automated daily regression testing

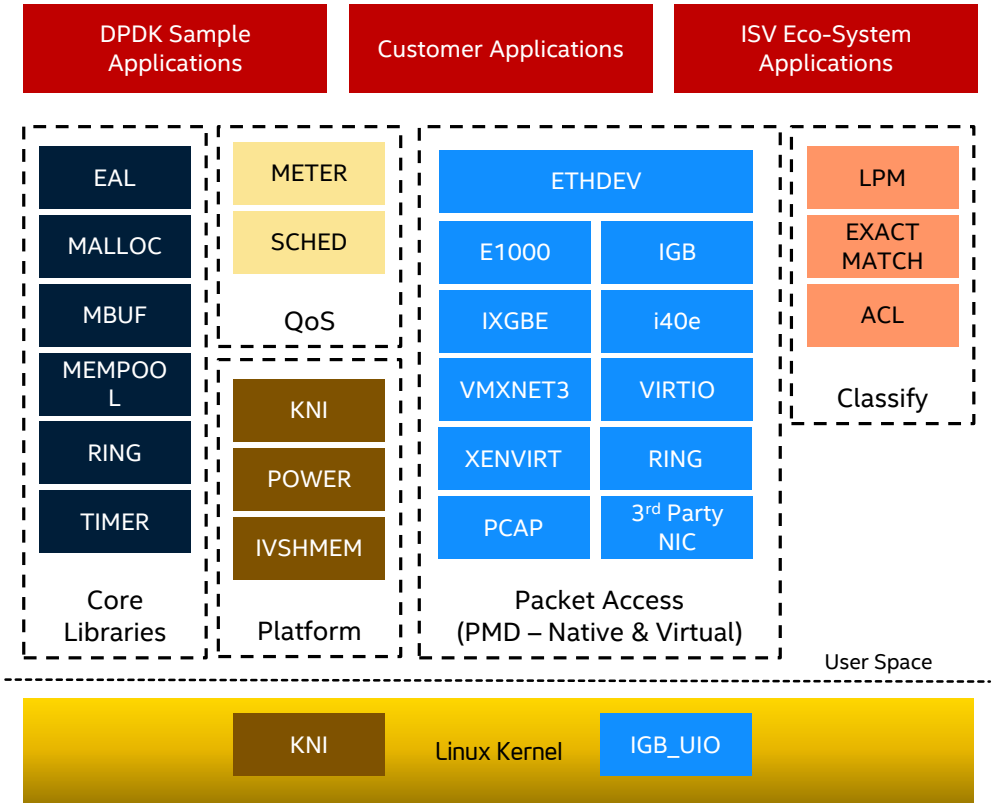
# Licensing

- DPDK is BSD licensed:
- http://opensource.org/licenses/BSD-3-Clause
- User is free to modify, copy and re-use code
- No need to provide source code in derived software (unlike GPL license)

# DPDK: a full open source community effort

- DPDK was officially launched on Sep 17, 2010 under the most liberal BSD open source license.
- Since April 2013, DPDK is available at www.dpdk.org as a fully independent, open source community.
- Many silicon suppliers have independently built and publically announced support for DPDK, such as Tilera, Netronome, Cavium and Xilinx.
- DPDK now available on Red Hat Enterprise Linux Extras channel

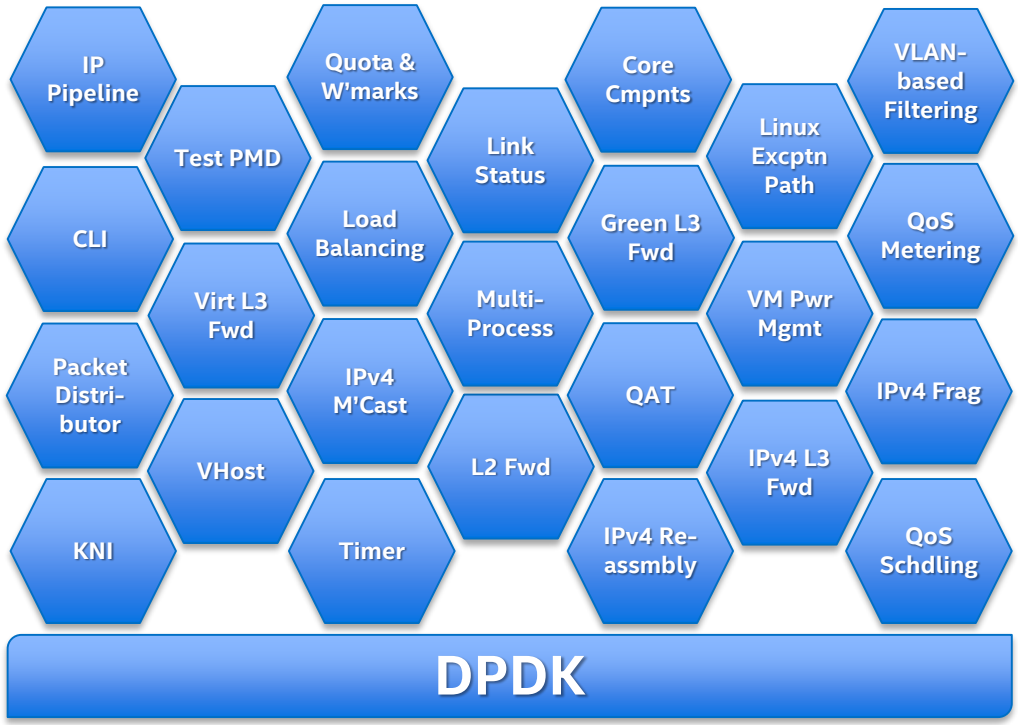
# Data Plane Development Kit




- Libraries for network application development on Intel Platforms
  - Speeds up networking functions
  - Enables user space application development
  - Facilitates both run-to-completion and pipeline models
- Free, Open-sourced, BSD Licensed
  - http://www.intel.com/go/dpdk
  - Git: http://dpdk.org/git/dpdk
- Scales from Intel Atom to multi-socket Intel Xeon architecture platforms
- About 30 pre-built example applications

# Build with DPDK

## Provided Sample Applications




- Over 30 pre-built sample applications
- Provide a great jump start for accelerating workloads with DPDK

# Intel Packet Processing Concepts

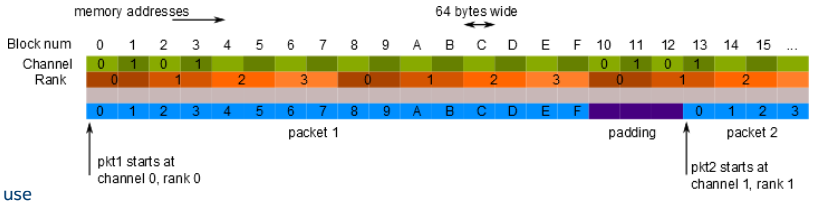
- DPDK is designed for high-speed packet processing on IA. This is achieved by optimizing the software libraries to IA with some of the following concepts
  - Huge Pages
  - Prefetching
  - Intel® DDIO
  - Cache alignment
  - New Instructions
  - Memory Interleave
  - Ptheads with Affinity
  - NUMA
  - Memory Channel

- Intel® Data Direct I/O Technology (Intel® DDIO)
  - Enabled by default in all Intel® Xeon® processor E5-based platforms
  - Enables PCIe adapters to route I/O traffic directly to L3 cache, reducing unnecessary trips to system memory, providing more than double the throughput of previous-generation servers, while further reducing power consumption and I/O latency.

- Pthreads
  - On startup of the DPDK specifies the cores to be used via the Pthread call with affinity to tie an application to a core. Reducing the kernel's ability of moving the application to another local or remote core affecting performance.
  - The user may still use Ptheads or Fork calls after the DPDK has started to allow threads to float or multiple thread to be tied to a single core.

- NUMA
  - DPDK utilizes NUMA memory for allocation of resources to improve performance for processing and PCIe I/O local to a processor.
  - With out the NUMA set in a dual socket system memory is interleaved between the two sockets.

- Huge Pages
  - DPDK utilizes 2M and 1G hugepages to reduce the case of TLB misses which can significantly affect a cores overall performance.




- Cache Alignment
  - Better performance by aligning structures on 64 Byte cache lines.

- Software Prefetching
  - needs to be issued "appropriately" ahead of time to be effective. Too early could cause eviction before use
  - Allows cache to be populated before data is accessed

- Memory channel use
  - Memory pools add padding to objects to ensure even use of memory channels
  - Number of channels specified at application start up

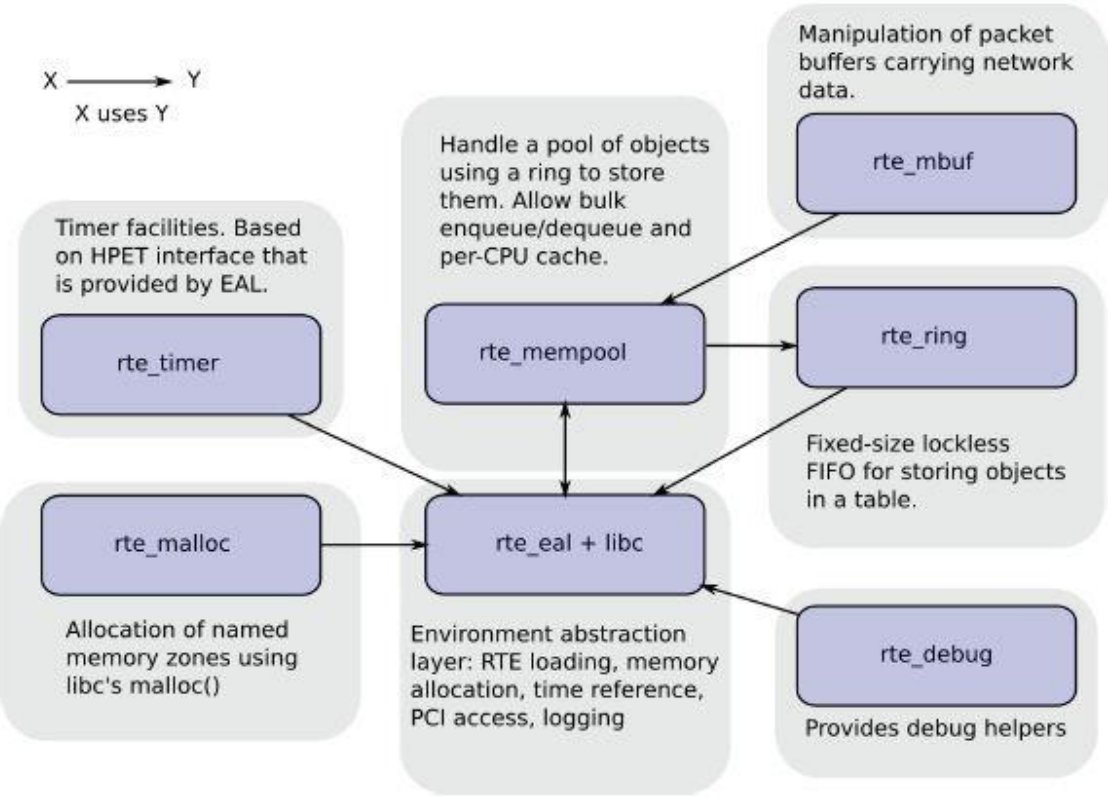
# The libraries/components (1)

| Library | |
|---|---|
| librte_eal | Environment Abstraction Layer. Meant to hide system/OS specifics from "common" upper layers |
| librte_malloc | rte_malloc() - replacement for malloc(). Allows allocation of data structures backed by huge pages |
| librte_mempool<br>librte_mbuf | Memory management: DPDK buffer pool management and packet buffer implementations |
| librte_ring | High speed ring for inter-core/process pointer passing |
| librte_timer | Timer routines |
| librte_lpm | Accelerated longest prefix match |
| librte_hash | Hash driven key-value exact match for tuple matching |
| librte_acl | Accelerated implementation of an Access Control List |

# The libraries/components (2)

| Library | |
|---|---|
| librte_meter | Meter/mark library: Implements srTCM (RFC 2697) and trTCM RFC 2698) |
| librte_sched | Hierarchical traffic shaper in software |
| librte_pmd* | Packet Access “Poll” mode drivers |
| librte_ether | Generic Ethernet device abstraction – the DPDK PMD API |
| librte_cmdline | Command line parser library |
| librte_distributor | A work queue distributor |
| librte_power | Power management primitives |
| librte_ivshmem | Shared memory implementation for inter-VM communication |
| KNI, librte_kni | Kernel Network Interface – implements a kernel netdev for passing packets into the kernel from DPDK |

# Relationship of DPDK Libraries



# The Buffer Management API (librte_mempool)

## The Buffer manager allocates memory from the EAL and creates pools with fixed element sizes

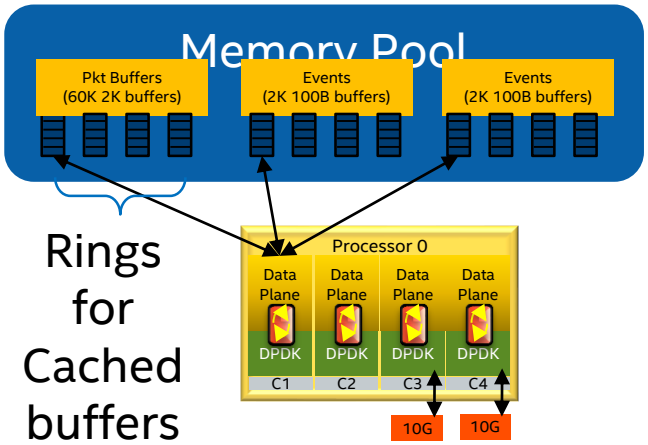
- Typical usage is packet buffers, descriptor ring buffers etc.
- Intent is to speed up runtime allocation/deallocation
- Does not support runtime resizing of pools

## Handles striping of buffers across a contiguous memory space

- Required to make sure we balance incoming packet load across all available memory channels

## Optimized for performance

- Cache alignment/page alignment
- Per core buffer caches for each buffer pool so that can allocate/deallocate without locks
- Bulk allocation/deallocation support



# Packet Buffer Management Structure (librte_mbuf)

## Concept is similar to Linux SKB or BSD mbuf

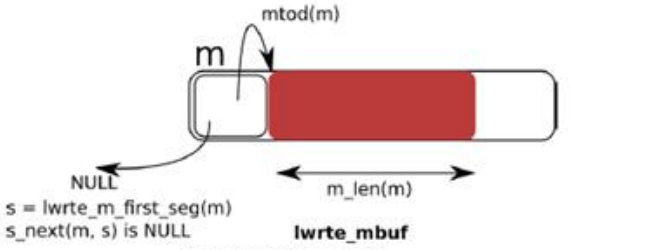
- Used to manage packet + metadata associated with the packet
- mbufs can be chained to provide a larger virtual buffer to transmit/receive jumbo packets

## Optimized for performance

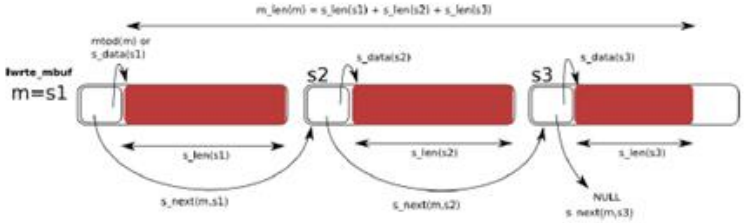
- Alignment, and structure of elements is taken care of
- Immediately precedes the packet buffer

## Will have macros/functions that allow operations

- To insert data at head or tail (pull/push)
- Chain manipulation etc.




*A mbuf with 1 segment*




*A mbuf with 3 segments*

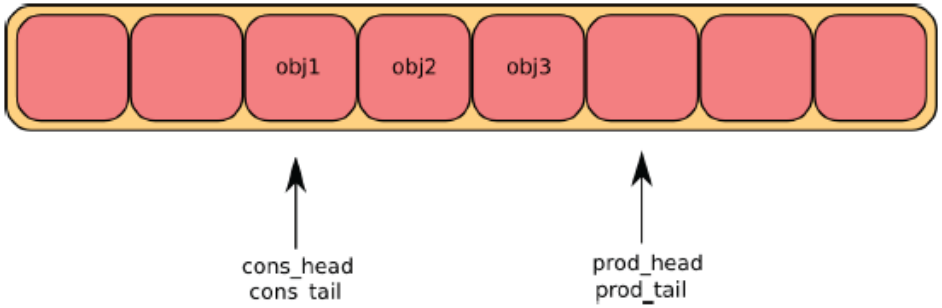
# Queue/Ring Management API (librte_ring)

## Effectively a FIFO implementation in software

- Lockless implementations for single or multi-producer, single consumer enqueue/dequeue
- Supports bulk enqueue/dequeue to support packet-bunching
- Implements high & low water mark thresholds for backpresssure/flow control

## Essential to optimizing for throughput

- Used to decouple stages of a pipeline (example later in slide-deck)



# DPDK Feature List

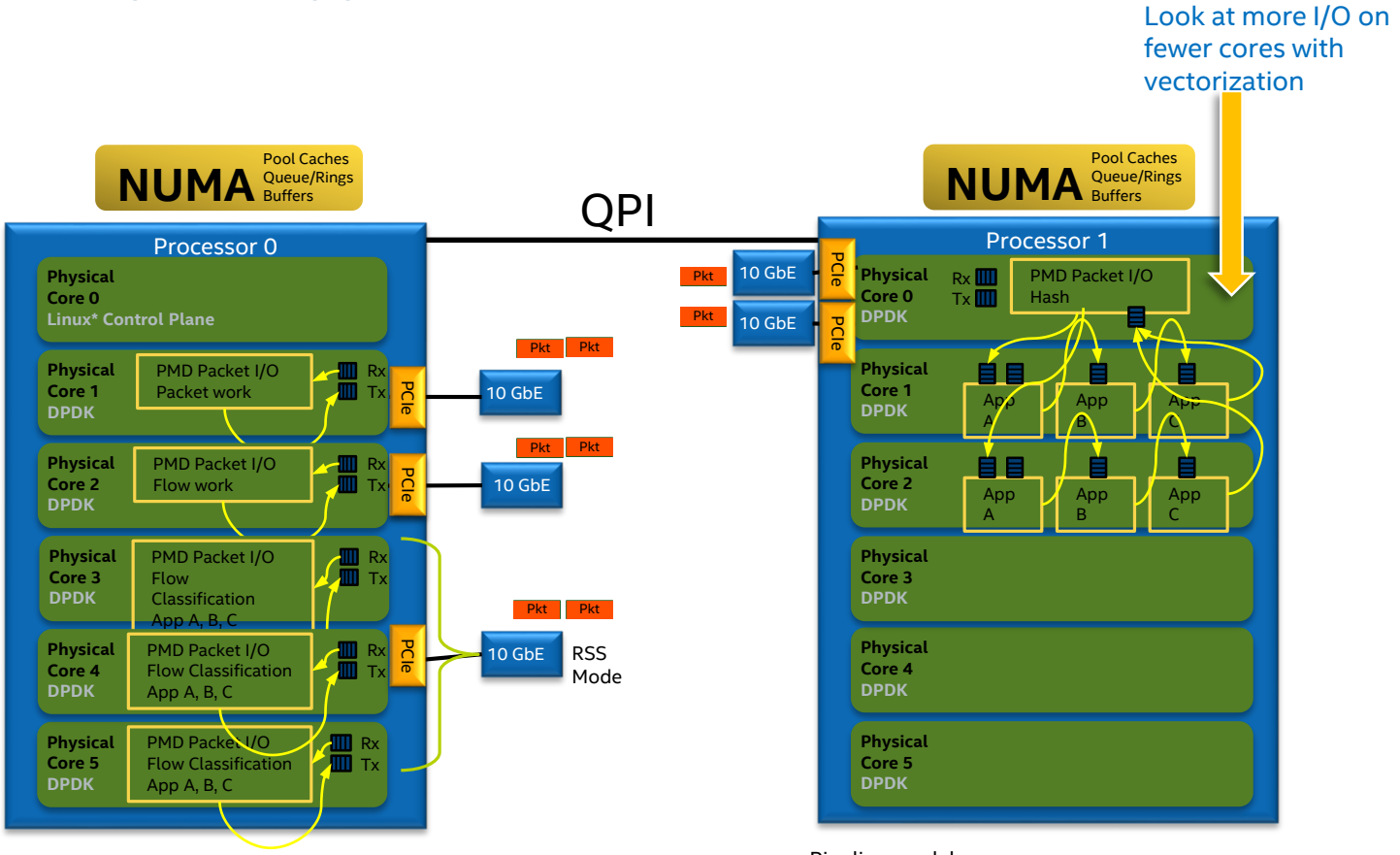
- Supported Operating Systems
  - Fedora release
  - Ubuntu*
  - Wind River* Linux*
  - Red Hat* Enterprise Linux
  - SUSE Enterprise Linux*
- Core components
  - rte_mempool: allocator for fixed-sized objects
  - rte_ring: single- or multi- consumer/producer queue implementation
  - rte_timer: implementation of timers
  - rte_malloc: malloc-like allocator
  - rte_mbuf: network packet buffers, including fragmented buffers
  - rte_hash: support for exact-match flow classification in software
  - rte_lpm: support for longest prefix match in software for IPv4 and IPv6
  - rte_sched: support for QoS scheduling
  - rte_meter: support for QoS traffic metering
  - rte_power: support for power management
- Environment Abstraction Layer (librte_eal)
  - Multi-process support
  - Multi-thread support
  - 1 Gbyte and 2 Mbyte page support
  - Atomic integer operations
  - Querying CPU support of specific features
  - High Precision Event Timer support (HPET)
  - PCI device enumeration and blacklisting
  - Spin locks and R/W locks
- Poll Mode Driver - Common (rte_ether)
  - VLAN support
  - Support for Receive Side Scaling (RSS)
  - IEEE1588
  - Buffer chaining; Jumbo frames
  - TX checksum calculation
  - Configuration of promiscuous mode, and multicast packet receive filtering
  - L2 Mac address filtering
  - Statistics recording
- Poll Mode Driver - 1 GbE Controllers (librte_pmd_e1000) support for
  - Intel® 82576 Gigabit Ethernet Controller (previously code named “Kawela”)
  - Intel® 82580 Gigabit Ethernet Controller (previously code named “Barton Hills”)
  - Intel® I350 Gigabit Ethernet Controller (previously code named “Powerville”)
  - Intel® 82574L Gigabit Ethernet Controller - Intel® Gigabit CT
  - Desktop Adapter (previously code named “Hartwell”)
  - Intel® Ethernet Controller I210 (previously code named “Springville”)

# DPDK Feature List Continued…

- Poll Mode Driver - 10 GbE Controllers (librte_pmd_ixgbe) support for
  - Intel® 82599 10 Gigabit Ethernet Controller (previously code named "Niantic")
  - Intel® Ethernet Server Adapter X520-T2 (previously code named "Iron Pond")
  - Intel® Ethernet Controller X540-T2 (previously code named "Twin Pond")
  - Virtual Machine Device Queues (VMDq) and Data Center Bridging (DCB) to divide incoming traffic into 128 RX queues. DCB is also supported for transmitting packets.
  - auto negotiation down to 1 Gb
  - Flow Director
- Quality of Service (QoS)
  - Hierarchical scheduler implementing 5-level scheduling hierarchy (port, subport, pipe, traffic class, queue) with 64K leaf nodes (packet queues).
  - Packet dropper based on Random Early Detection (RED) congestion control mechanism.
  - Traffic Metering based on Single Rate Three Color Marker (srTCM) and Two Rate Three Color Marker (trTCM).
  - Quality of Service (QoS) Hierarchical Scheduler: Subport Traffic Class Oversubscription
- Virtualization (KVM)
  - Para-virtualization supports
    - virtio front-end poll mode driver in guest virtual machine
    - vHost raw socket interface as virtio back-end via KNI
  - SR-IOV Switching for the 10G Ethernet Controller supports
    - Support Physical Function to start/stop Virtual Function Traffic
    - Support Traffic Mirroring (Pool, VLAN, Uplink and Downlink)
    - Support VF multiple MAC addresses (Exact/Hash match), VLAN filtering
    - Support VF receive mode configuration
- Miscellaneous
  - New libpcap-based poll-mode driver, including support for reading from 3rd Party NICs using Linux kernel drivers
  - Support for building the DPDK as a shared library
  - Support for multiple instances of the DPDK
  - Multi-thread Kernel NIC Interface (KNI) for interaction with kernel

# PCIe* connectivity and core usage

## Using run-to-completion or pipeline software models




Run to Completion model
- I/O and Application workload can be handled on a single core
- I/O can be scaled over multiple cores

Pipeline model
- I/O application disperses packets to other cores
- Application work performed on other cores

# When to Choose Run-to-Completion vs. Pipeline

Applications will generally employ both models

Technical questions to consider:

- How many cycles/packet do I need for my algorithms?
- Are there large data structures that need to be shared with read/write access across packets?
- Will I support timer / packet ordering functions?
- Can I take advantage of a specific optimization if you restrict an algorithm to one core?
- How much data would I need to exchange between software modules?

More...

# More Run-to-Completion vs. Pipeline…

General architecture questions to consider:

- Do some cores have easier/faster access to a hw resource?
- Do you want to view cores as offload engines?

Development environment questions to consider:

- Do you need to employ legacy software modules?
- Does ease-of-code-maintenance trump performance?

# NUMA

Non Uniform Memory Access

# Non-Uniform Memory Access (NUMA)

FSB architecture (legacy)

- All memory in one location

Starting with Intel® Core™ microarchitecture (Nehalem)

- Memory located in multiple places

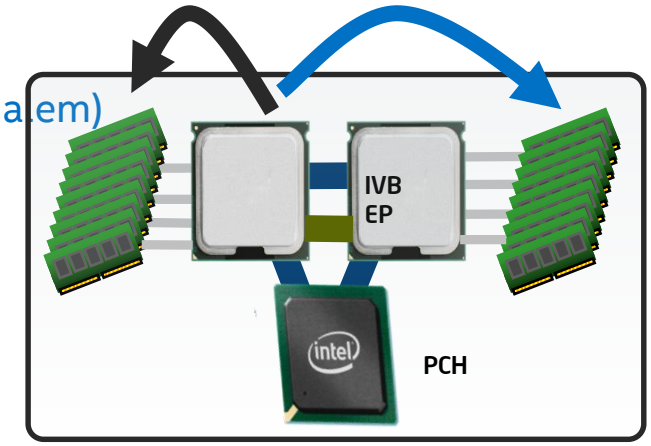
Latency to memory dependent on location

Local memory

- Highest BW
- Lowest latency

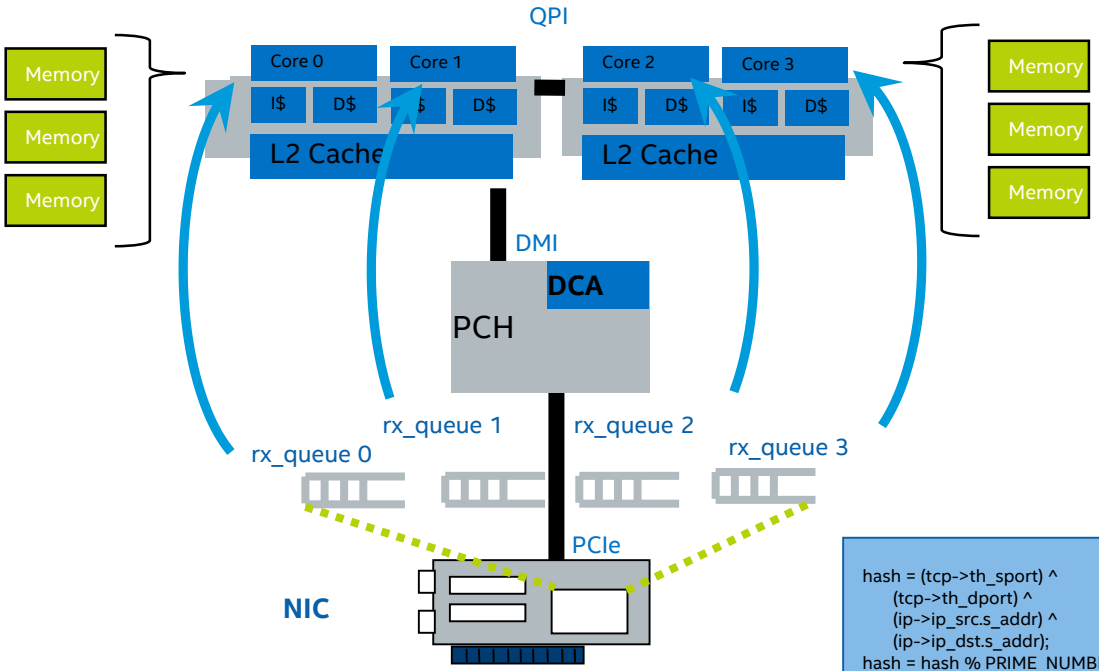
Remote Memory

- Higher latency




*Ensure software is NUMA-optimized for best performance*

# NUMA Considerations for Data Structure Allocation




```
hash = (tcp->th_sport) ^
       (tcp->th_dport) ^
       (ip->ip_src.s_addr) ^
       (ip->ip_dst.s_addr);
hash = hash % PRIME_NUMBER;
return lookup_table[hash];
```

PTU Metrics

- MEM_UNCORE_RETIRED.REMOTE_DRAM
- MEM_INSTRUCTIONS_RETIRED.LATENCY_ABOVE_THRESHOLD

# CACHING ON INTEL ARCHITECTURE

# Caching on IA

- IA Processors have cache integrated on processor die.
  - Fast access SRAM
  - Code & data from system memory (DRAM) stored in fast access cache memory
- Without a cache – CPU runs out of instructions from system memory
  - CPU Core “stalls” – waiting for data
- Cache miss (data not in cache)
  - CPU needs to get data from system memory
  - Cache populated with required data
  - Not just the data required, but a block of info is copied
    - “Cache line” – 64 Bytes on IA (IVB, HSW etc.)
- Cache hit – data present in cache

# Caching on IA

- What can be cached?
  - Only DRAM can be cached
  - IO, MMIO never cached
- L1 cache is smallest, and fastest.
- L1 Code cache is read-only
- Address residing in L1/L2 must be present in L3 cache – “inclusive cache”

# Translation Lookaside Buffers (TLBs)

- TLBs – Translation Lookaside Buffers – 2 types
  - Instruction TLB
  - Data TLB
- TLB is cache – maps virtual memory to physical memory
  - When memory requested by application, OS maps virtual address from process to physical address in memory
  - Mapping of virtual to physical memory – Page Table Entry (PTE)
  - TLB is a cache for the Page Table
  - If data is found in TLB during address lookup
    - TLB hit
    - Otherwise – TLB miss (page walk) - performance hit
    - Huge pages (Linux) – can alleviate

# Translation Lookaside Buffers (TLBs)

- TLBs are a cache for page tables
- If memory address lookup is not in TLB -> TLB miss
  - We must then “walk the page tables”
  - This is slow, and costly
- We need to minimise TLB misses
- Solution is to use huge pages
  - Use 2M or 1G huge pages instead of default 4k pages

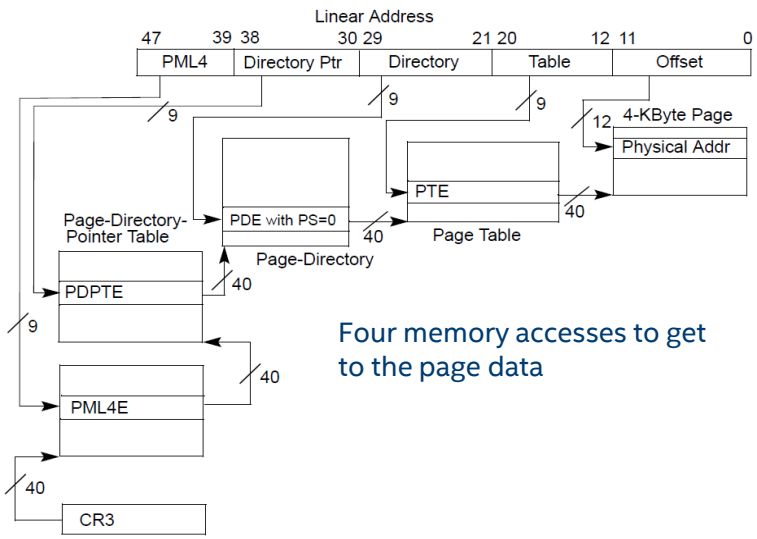
# HUGE PAGES

# Huge Pages

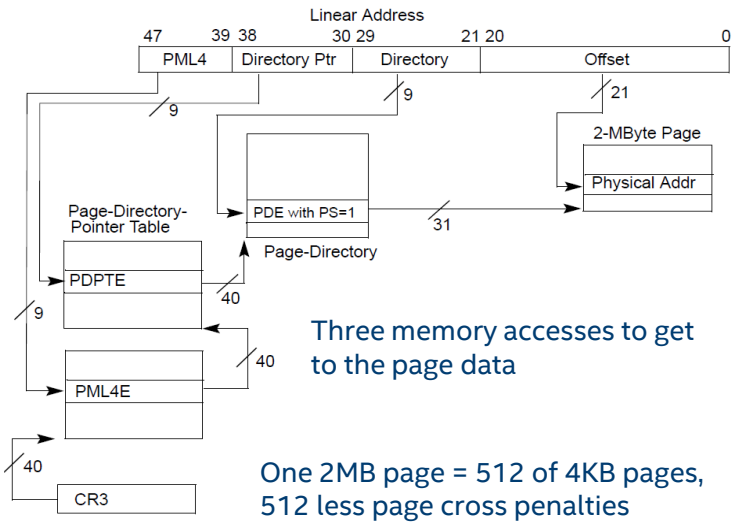
- All memory addresses virtual
  - Memory appears contiguous to applications, even if physically fragmented
- Map virtual address to physical address
  - Use page tables to translate virtual address to physical address
  - Default page size in Linux on IA is 4kb.
  - 4 layers of page tables

# Why Hugepages?

TLB maps page numbers to page frames. Each TLB miss requires page walk.




Four memory accesses to get to the page data




Three memory accesses to get to the page data

One 2MB page = 512 of 4KB pages, 512 less page cross penalties

DTLB:

- 4K pages 64 entries, maps 256 KB, so to access 16G of memory 32MB of PTE tables read by CPU
- 2M pages 32 entries, maps 64 MB, so to access 16G of memory 64Kb of PDE tables read by CPU, fits into CPU cache

# Huge Pages

- Use Linux hugepage support through “hugetlbfs” filesystem
- Each page is 2MB in size equivalent to 512 4KB pages
- Each page requires only 1 DTLB entry
- Reduce DTLB misses, and therefore page walks
- Gives improved performance
- Need to enable & allocate huge pages with Linux boot command (in GRUB file)
  - Better to enable at boot time – prevents fragmentation in physical memory

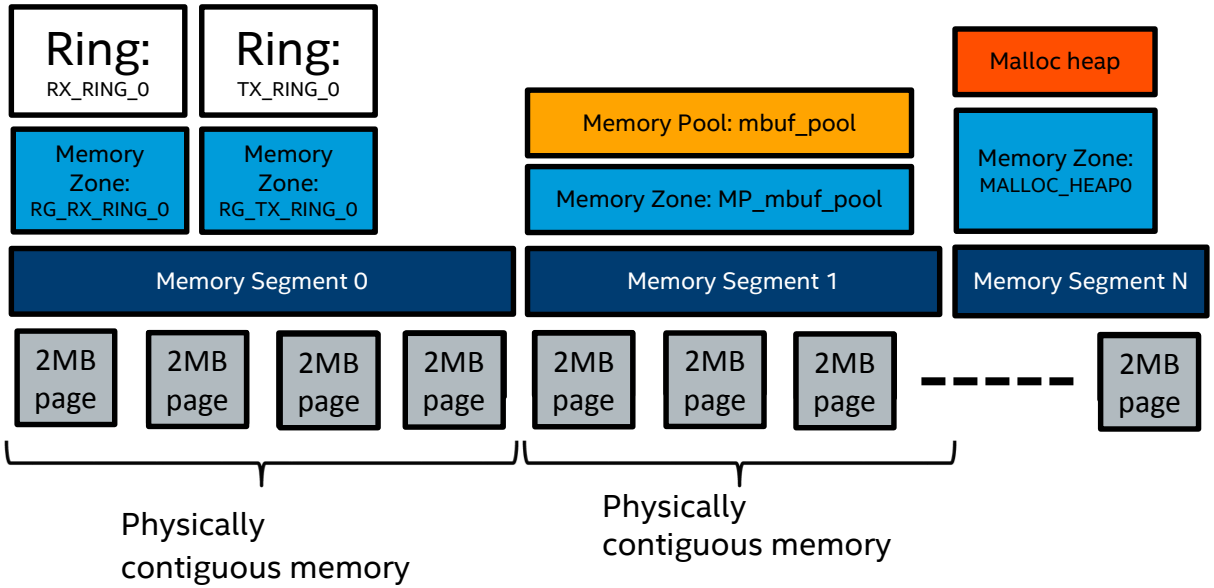
# MEMORY CONFIGURATION

DPDK

# Memory allocation

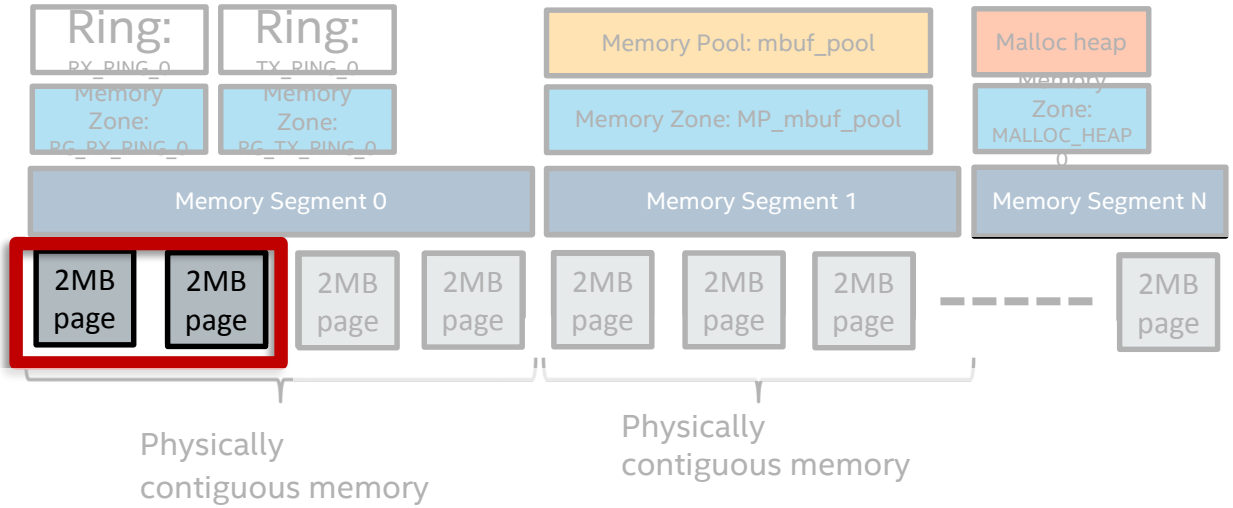
- For DPDK application – allocated all memory from huge pages
- Allocate all memory at initialisation time (not during run time).
- Pools of buffers created.
  - Buffers taken from pools as needed for packet processing
  - Returned to pool after use
  - Never need to use “malloc” at runtime.
  - DPDK takes care of aligning memory to cache lines

# Memory Object Hierarchy



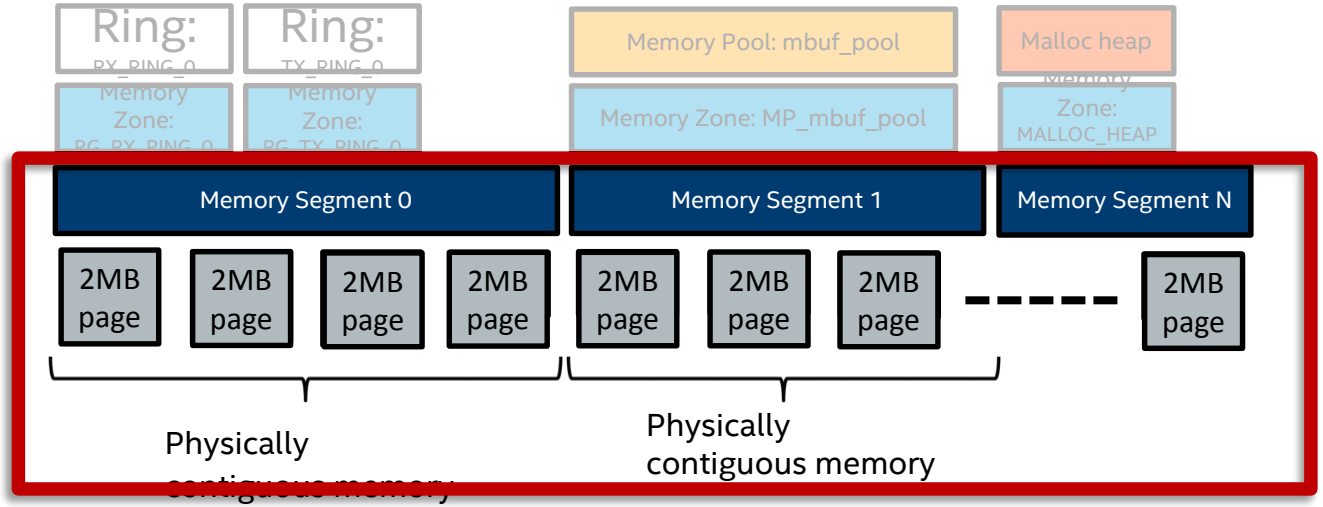
# Hugepages

- Use Linux hugepage support through “hugetlbfs” filesystem
- Each page is 2MB in size equivalent to 512 4KB pages
- Each page requires only 1 DTLB entry
- Reduce DTLB misses, and therefore page walks
- Gives improved performance



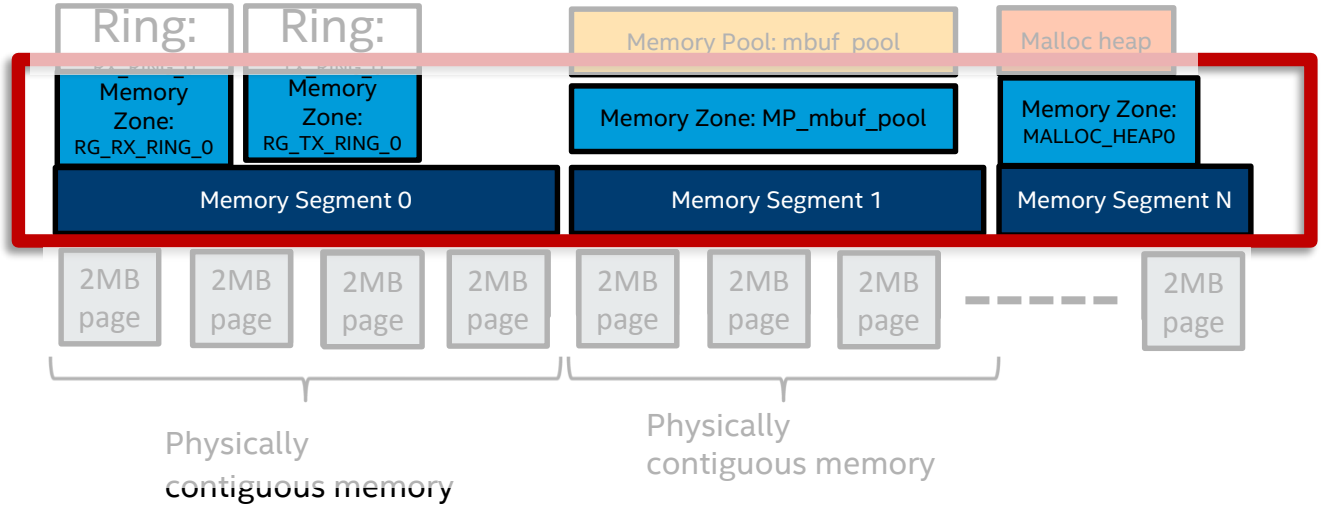
# Memory Segments

- Internal unit for memory management is the memory segment
- Always backed by Huge Page (2 MB/1 GB page) memory
- Each segment is contiguous in physical and virtual memory
- Broken out into smaller memory zones for individual objects



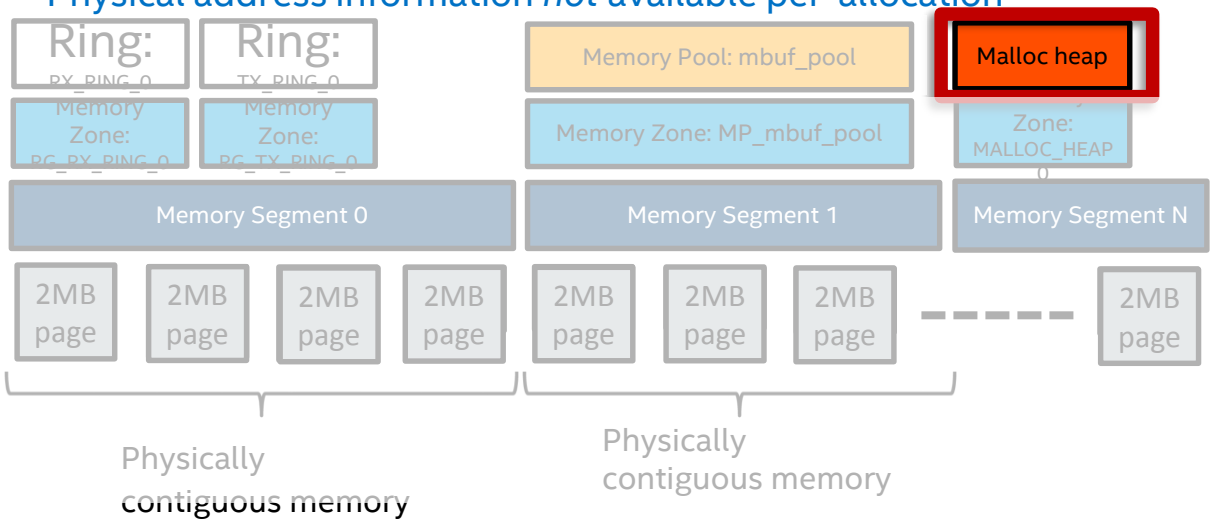
# Memory Zones

- Most basic unit of memory allocation – named block of memory
- Allocate-only, cannot free
- Cannot span a segment boundary – contiguous memory
- Physical address of allocated block available to caller



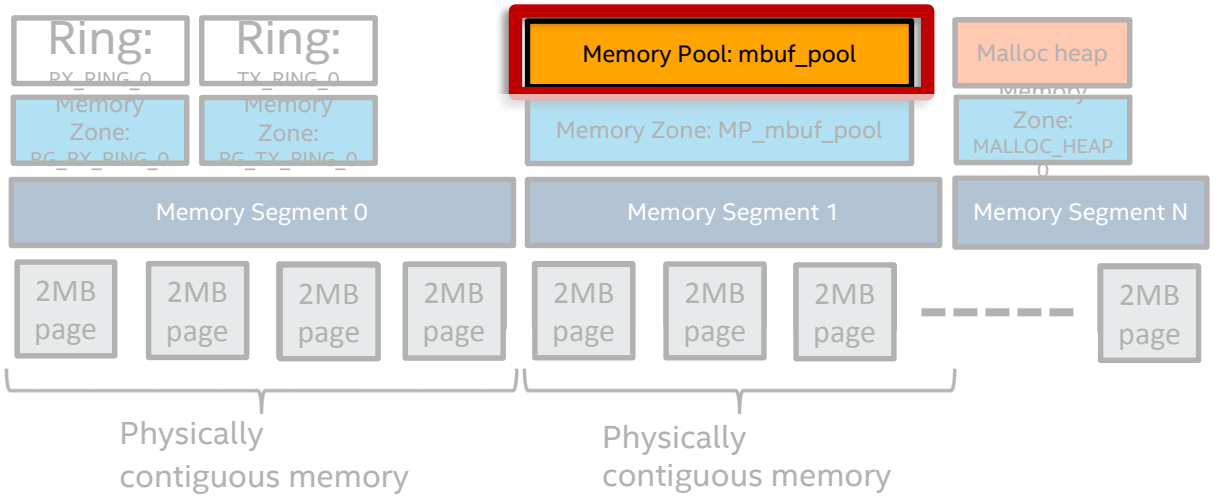
# Malloc support – rte_malloc/rte_free

- Malloc library provided to allow easier application porting
- Backed by one or more memzones
- Uses hugepage memory, but supports memory freeing
- Not lock-free – avoid in data path
- Physical address information *not* available per-allocation



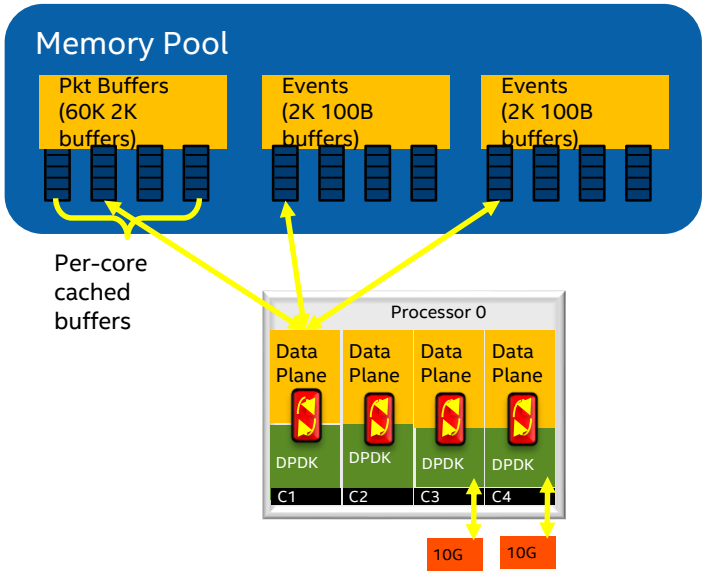
# Memory Pools

- Pool of fixed-size buffers
- One pool can be safely shared among many threads
- Lock-free allocation and freeing of buffers to/from pool
- Designed for fast-path use



# Memory Pools (continued)

- **Size fixed at creation time:**
  - Fixed size elements
  - Fixed number of elements
- **Multi-producer / multi-consumer safe**
- **Safe for fast-path use**
- **Typical usage is packet buffers**
- **Optimized for performance:**
  - No locking, use CAS instructions
  - All objects cache aligned
  - Per core caches to minimise contention / use of CAS instructions
  - Support for bulk allocation / freeing of buffers



# Memory allocation

- rte_eal_init()
  - Initialises Environment Abstraction Layer
  - Takes care of allocating memory from huge pages
- rte_mempool_create()
  - Create pool of message buffers (mbufs)
  - This pool is used to hold packet data
  - mbufs taken from and returned to this pool

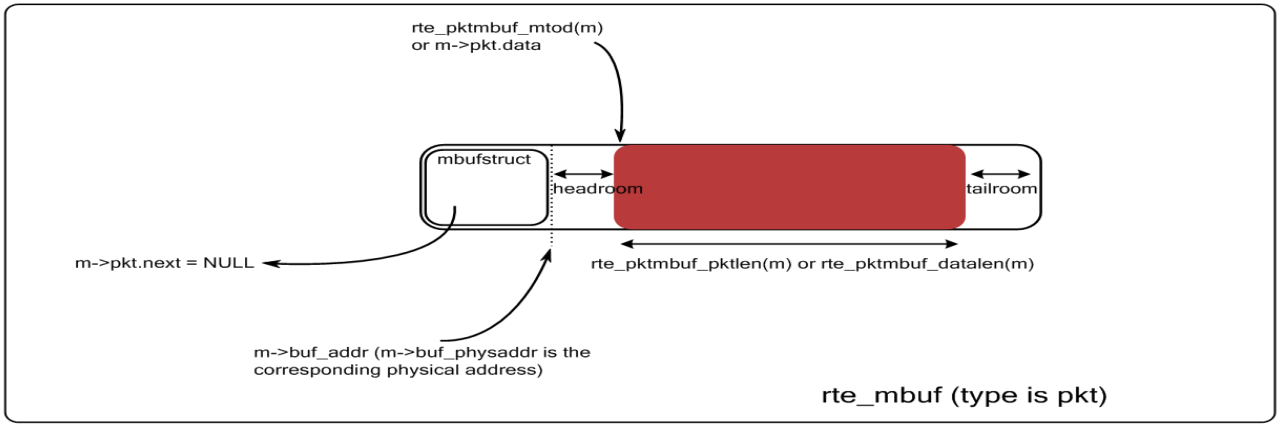
# Memory Buffer - mbuf

Memory buffer structure used throughout DPDK

Header holds meta-data about packet and buffer

- Buffer & packet length
- Buffer physical address
- RSS hash or flow director filter information
- Offload flags

Body holds packet data plus room for additional headers and footers.

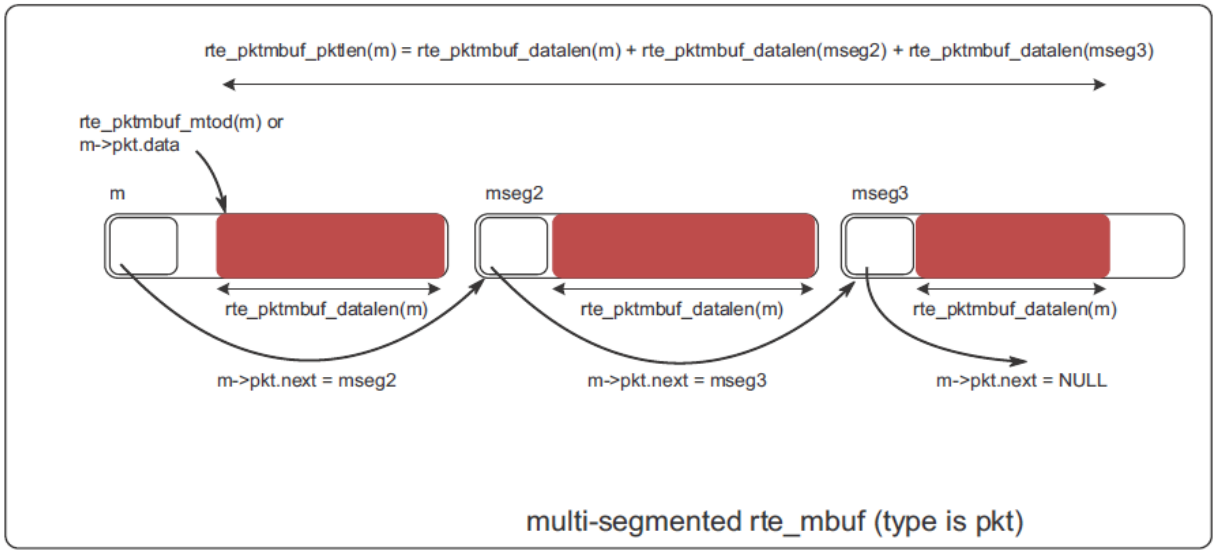


# Memory Buffer – chained mbuf

Mbufs generally used with memory pools

Size of mbuf fixed when the mempool is created

For packets too big for a single mbuf, the mbufs can be linked together in an "mbuf chain"



# DPDK Versions

## DPDK version 2.1 currently available on www.dpdk.org

## Some of the new Features in 2.1:

- Cuckoo Hash + Updated Jhash.
- IEEE1588 Support
- PCI Hot Plug – PMD Support
  - e1000 hotplug
  - ixgbe hotplug
  - i40e hotplug
  - fm10k hotplug
- Packet Framework Enhancements
  - New configuration file syntax
  - New implementation of pass-through pipeline, firewall pipeline, routing pipeline, and flow classification
  - Master pipeline with CLI interface
- I40e: Mirroring Rule
- I40e: Double VLAN Strip/Insert
- I40e: Unified Packet Type
- I40e Flow Director (L2_payload Type and VF Filtering)
- VXLAN Offload Sample Application
- Extended NIC statistics
- Dynamic Memzone
- Red Rock Canyon (FM10K) Features
  - fm10k promiscuous
  - mac vlan filtering,
  - Tx checksum offload)
- Interrupt Mode
- Cisco Ethtool (excl. sample app)

## Watch out for:

- DPDK 2.2 Coming November 2015
- Further BrightTalks in this series on DPDK
- DPDK Userspace 2015 Summit – Dublin, Ireland – October 8-9th 2015

intel®