# EC 440 – Introduction to Operating Systems

**Orran Krieger (BU)**
**Larry Woodman (Red Hat)**

# Purpose of Memory Management

- Providing a memory abstraction that is conducive to running(and debugging) programs
  - the virtual address space
- Creating and managing a separate private address space for each process.
- Multiplexing and managing all of the physical memory.
- Manage the memory hierarchy
  - memory to storage

# Address Space Overview

- Physical address space
  – processes share a flat physical memory layout
  – no protection, fixing up addresses...

- Virtual address space
  – per-process separate/private address space starting at zero ending at some high address.
  – Segmentation where large virtually contiguous regions map to large physical regions
    - Compaction, Swapping, external fragmentation
  – Rest of the time on Paging
    - Split into small discrete pages and maps many small discontiguous physical pages.

# Virtual Memory Management

- System calls to allocate and deallocate valid regions in the virtual address space.
  - mmap(), brk(), sbrk(), malloc()
  - munmap(), brk/sbrk, free()
- Two types of virtual memory regions
  - mapped file
  - anonymous

# Physical Memory Management

- Page faults and allocating memory to resolve virtual addresses
- Reclaiming physical memory when it becomes exhausted, several algorithms, local vs global.
  - LRU, second chance, working sets, etc.
- Page size issues (fragmentation) and overhead
- Using physical memory to cache filesystem data
  - aggressively allocating and reclaiming to minimize filesystem I/O
- Memory for kernel text and data structures, generally not reclaimed:
  - only released when memory freed, e.g., page table memory freed when the process exits

# Today

## Putting it in context with a real system: i.e. Linux on x86-64

# In particular

- Page table structure
- Allocate and de-fragment physical memory
- How do we find out where a page is mapped
- A real buffer cache
- How we reclaim memory
- How a page fault works

Bonus martial; the real pain in the butt NUMA

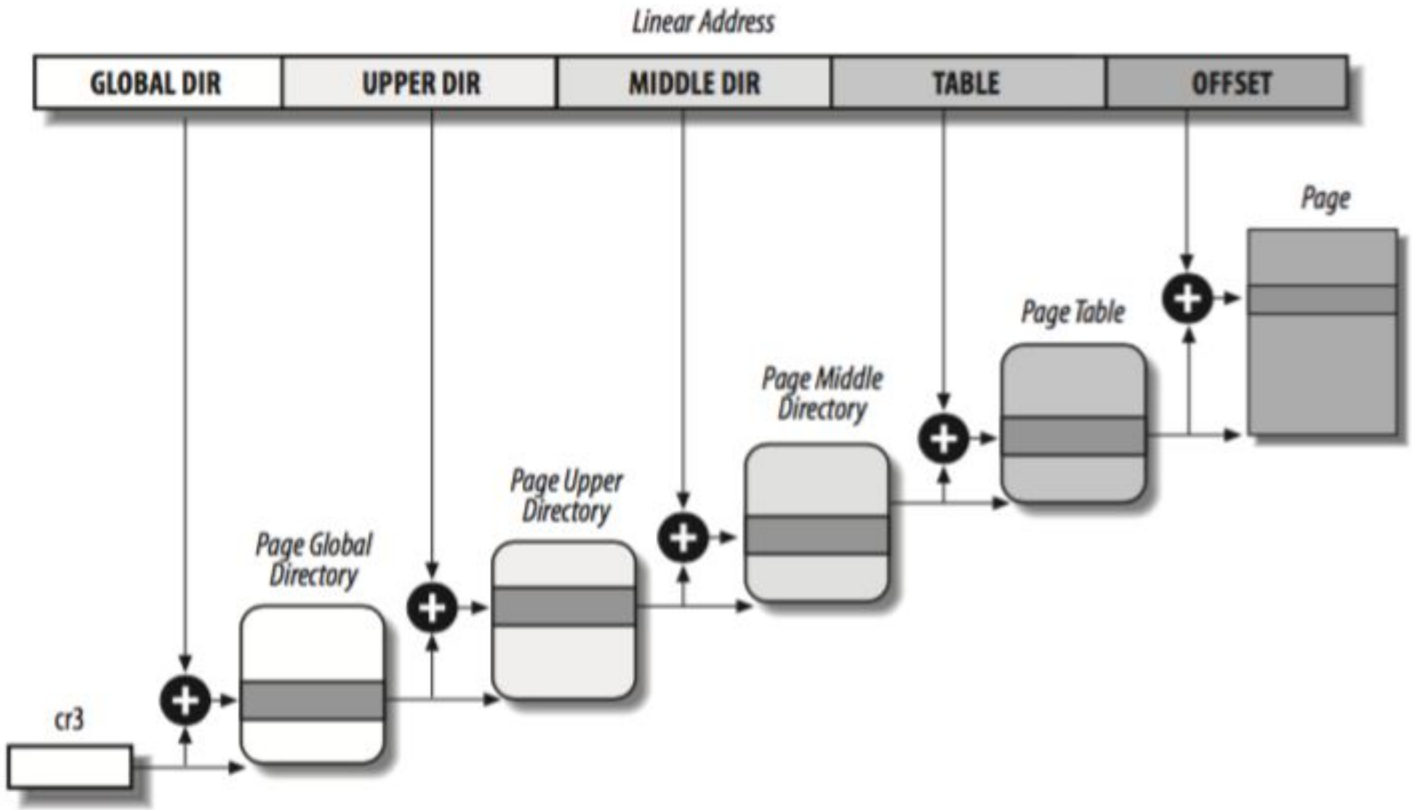
# Page tables in x86

# Page Tables

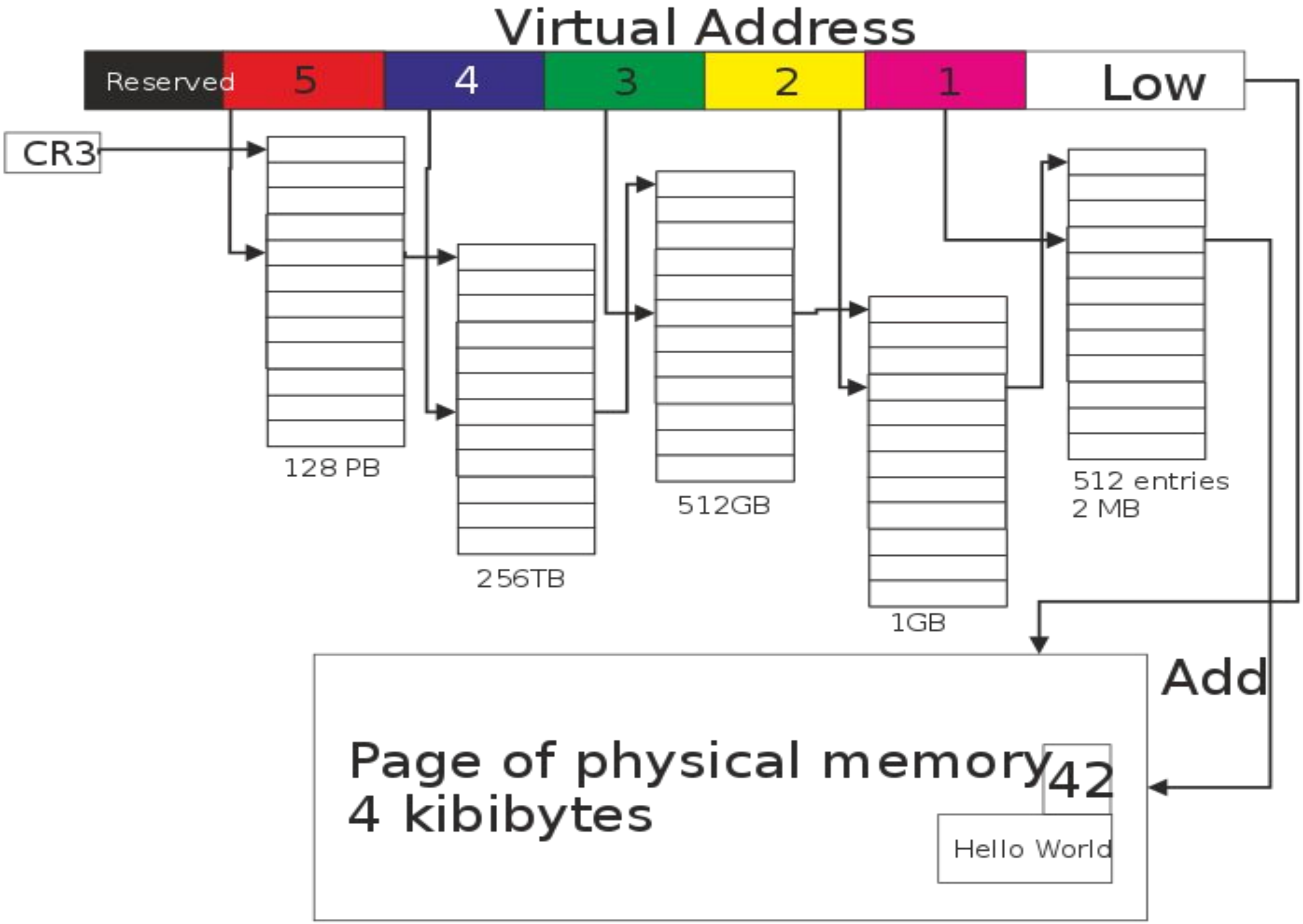
- Linux supports 2 Virtual Address space sizes and therefore 2 Page Tables layouts:
  - 256TB or 48 bits which required a 4 level page table
    - supports 64TB of RAM
  - 128PB or 57 bits which requires a 5-level page table
    - supports 32PB of RAM

- Selects which page table to use based on:
  - hardware that the Linux kernel is running on
  - amount of physical memory that is present

# Intel x86_64 4-level page tables

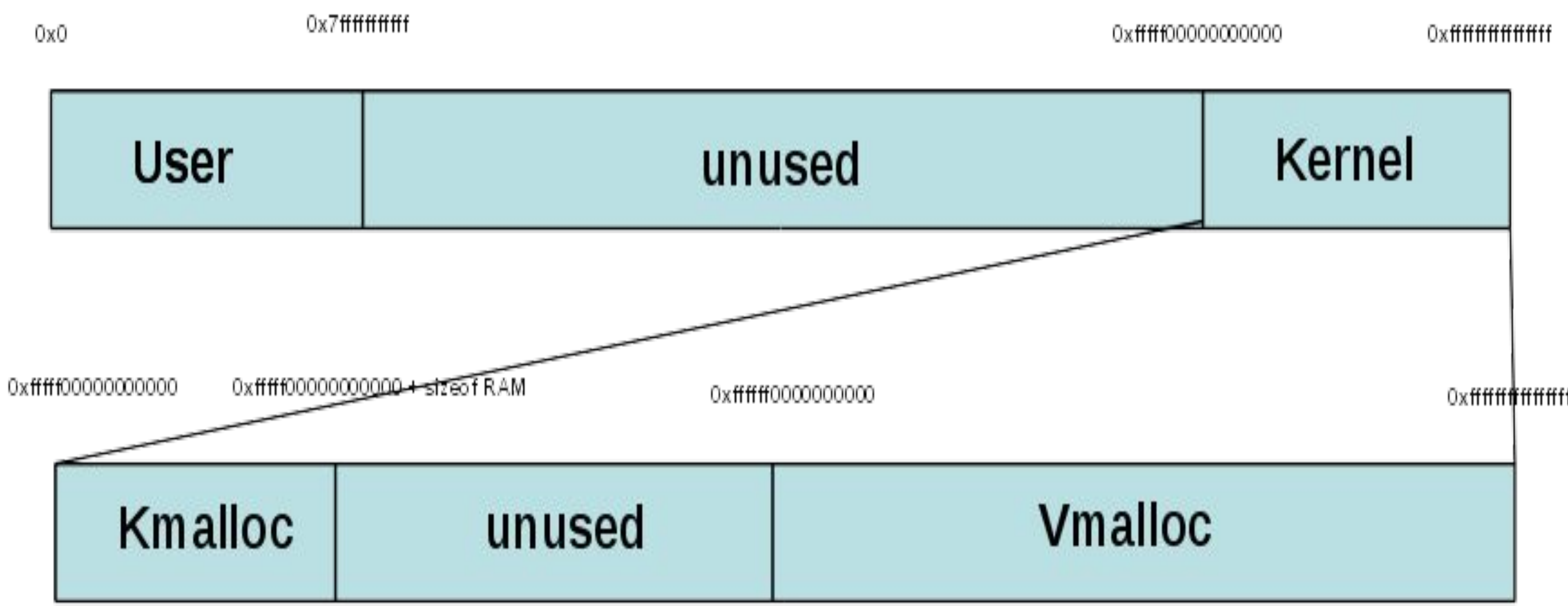


# Intel x86_64 5-level page tables



# The virtual map

# Linux Virtual memory layout



# Linux Virtual Memory Management

**User:** Virtual size = 2^47/128TB (with 48bit)
0x0 - 0x7fffffffffff(2^47)

**Kernel:** Virtual size = 2^47/128TB
0xffff000000000000 - 0xffffffffffffffff
**Kmalloc:** lower ½ of kernel(2^46 or 64TB)
physical memory directly mapped here.

**Vmalloc:** upper ½ of kernel(2^46 or 64TB)
Modules are loaded here.

**Unused:** Giant gap between User-End & Kernel-Start

# Kernel Address Space

- Most kernel memory is physically contiguous:
  - kmalloc region - used for allocating powers of 2 pages of physically contiguous memory
  - 4KB to 4MB
  - mapping uses 2MB pages

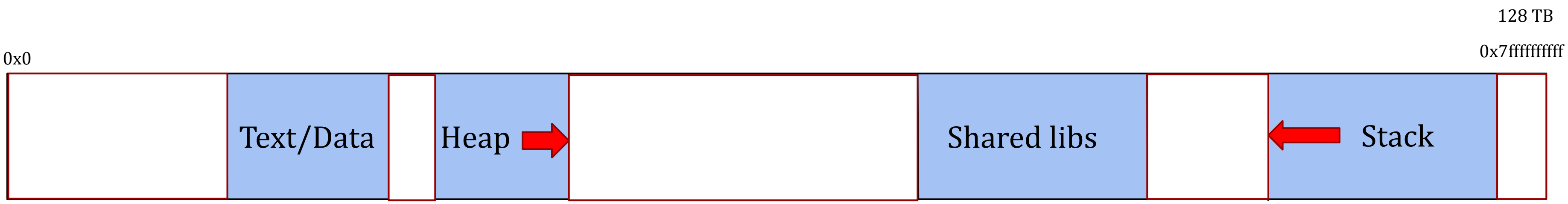
- We sometimes need huge areas that cannot be physically contiguous: e.g., modules
  - vmalloc region - used for allocated large virtual memory regions in the kernel
  - 4KB to 64TB
  - uses 4Kb mappings
  - mappings created on demand

# (review)2 Types of User Virtual Memory

- Mapped File Virtual Memory
  - A file that is mapped into the virtual address space
  - Created via mmap(fd=FD) system call.
  - Destroyed via munmap() system call

- Anonymous Virtual Memory
  - Any virtual memory not backed by a file(stack, heap, uninitialized data, etc.)
  - Created via mmap(fd=NULL), sbrk(), brk(), malloc()
  - Destroyed, via munmap(), sbrk(), brk(), free(), exit()

# Linux user virtual address space

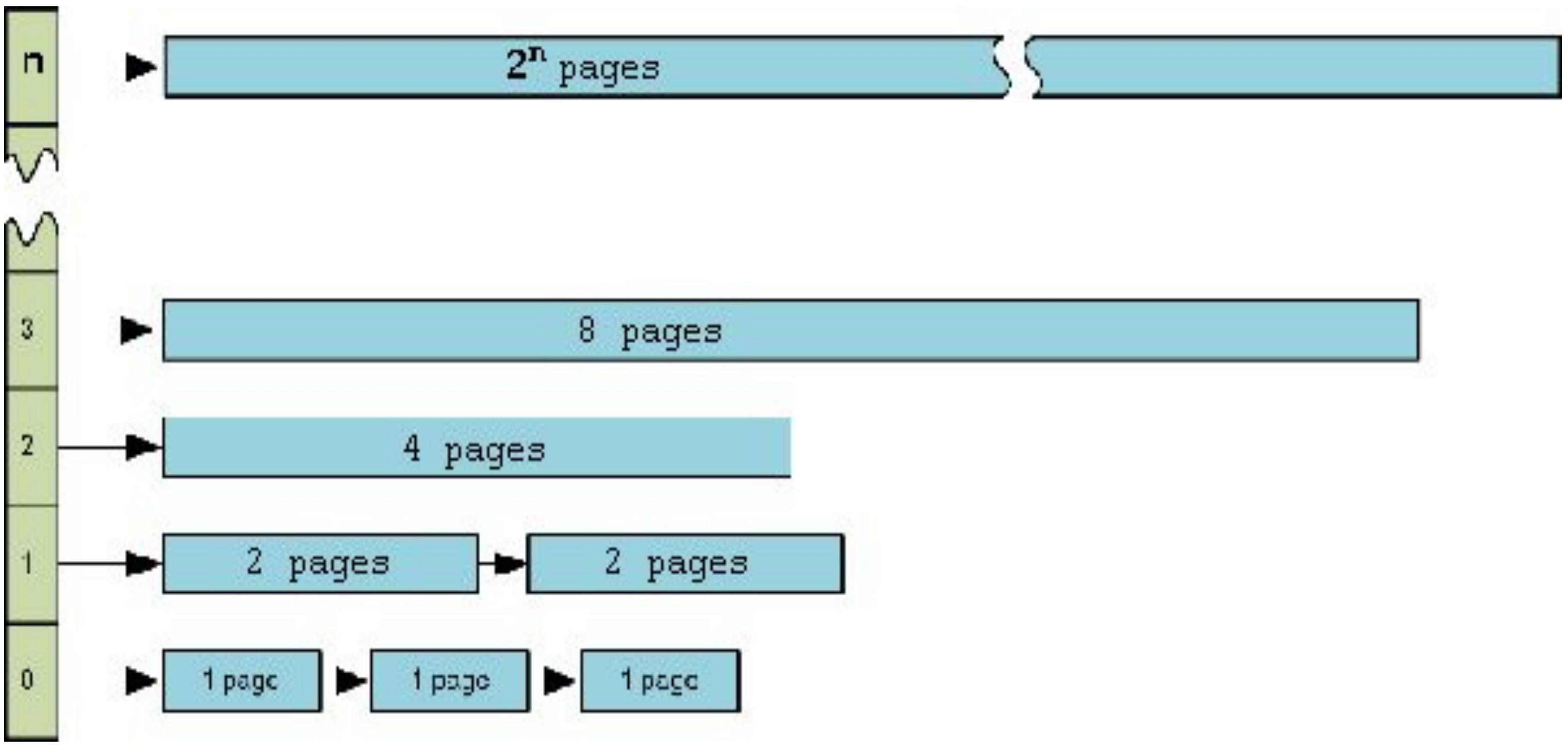



```
[lwoodman@lwoodman linux]$ cat /proc/self/maps
55a31c484000-55a31c486000 r--p 00000000 fd:00 3801429                    /usr/bin/cat
55a31c486000-55a31c48a000 r-xp 00002000 fd:00 3801429                    /usr/bin/cat
55a31c48a000-55a31c48c000 r--p 00006000 fd:00 3801429                    /usr/bin/cat
55a31c48c000-55a31c48d000 r--p 00007000 fd:00 3801429                    /usr/bin/cat
55a31c48d000-55a31c48e000 rw-p 00008000 fd:00 3801429                    /usr/bin/cat
55a31cdb2000-55a31cdd3000 rw-p 00000000 00:00 0                          [heap]
7f2b31d8c000-7f2b31dae000 rw-p 00000000 00:00 0
7f2b31dae000-7f2b3f2de000 r--p 00000000 fd:00 3806989                    /usr/lib/locale/locale-archive
7f2b3f2de000-7f2b3f2e0000 rw-p 00000000 00:00 0
7f2b3f2e0000-7f2b3f306000 r--p 00000000 fd:00 3813237                    /usr/lib64/libc-2.32.so
7f2b3f306000-7f2b3f455000 r-xp 00026000 fd:00 3813237                    /usr/lib64/libc-2.32.so
7f2b3f455000-7f2b3f4a0000 r--p 00175000 fd:00 3813237                    /usr/lib64/libc-2.32.so
7f2b3f4a0000-7f2b3f4a1000 ---p 001c0000 fd:00 3813237                    /usr/lib64/libc-2.32.so
7f2b3f4a1000-7f2b3f4a4000 r--p 001c0000 fd:00 3813237                    /usr/lib64/libc-2.32.so
7f2b3f4a4000-7f2b3f4a7000 rw-p 001c3000 fd:00 3813237                    /usr/lib64/libc-2.32.so
7f2b3f4a7000-7f2b3f4ad000 rw-p 00000000 00:00 0
7f2b3f4c4000-7f2b3f4c5000 r--p 00000000 fd:00 3815120                    /usr/lib64/ld-2.32.so
7f2b3f4c5000-7f2b3f4e6000 r-xp 00001000 fd:00 3815120                    /usr/lib64/ld-2.32.so
7f2b3f4e6000-7f2b3f4ef000 r--p 00022000 fd:00 3815120                    /usr/lib64/ld-2.32.so
7f2b3f4ef000-7f2b3f4f0000 r--p 0002a000 fd:00 3815120                    /usr/lib64/ld-2.32.so
7f2b3f4f0000-7f2b3f4f2000 rw-p 0002b000 fd:00 3815120                    /usr/lib64/ld-2.32.so
7ffcb575a000-7ffcb577b000 rw-p 00000000 00:00 0                          [stack]
7ffcb577d000-7ffcb5781000 r--p 00000000 00:00 0                          [vvar]
7ffcb5781000-7ffcb5783000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
[lwoodman@lwoodman linux]$
```

# How do we de-fragment?

# Defragment: Buddy Allocator

A power-of-2 page allocator that is used to allocate physically contiguous memory



# Buddy Allocator

- Automatically splits up larger chunks of memory if necessary
- Automatically defragments when when pages are freed
  - if neighbor chunk is freed coalesce together and create larger memory chunk
  - Uses the page struct at front of chunk to see if free at same granularity
- Note: fragmentation will still occur over time... so

# Defragmenting: Compactor

- creates large physically contiguous free chunks.
- slides in-use chunks of memory together.
  - same as compaction with segmentation
- these large chunks are coalesced together by the buddy allocator so kmalloc() will succeed
- runs periodically when lack of large chunks of free memory:
  - very expensive has to copy memory around

# Linux Physical Memory Management

struct page: one for each page of RAM
mem_map: array of struct pages
buddy page allocator: pages coalesced from 4KB to 4MB

- 11 buckets of physically contiguous RAM
- every allocation can split up a large bucket/page
- every free tries to coalesce into largest bucket

**DMA: 0*4kB 2*8kB 1*16kB 2*32kB 2*64kB 0*128kB 2*256kB 1*512kB 1*1024kB 1*2048kB 1*4096kB = 8416kB**
**DMA32: 3997*4kB 3042*8kB 2544*16kB 1995*32kB 1033*64kB 232*128kB 137*256kB 102*512kB 68*1024kB 1*2048kB 0*4096kB = 399652kB**
**Normal: 3997*4kB 3042*8kB 2544*16kB 1995*32kB 1033*64kB 232*128kB 137*256kB 102*512kB 68*1024kB 1*2048kB 0*4096kB = 399652kB**

- All RAM is freed/put free list at boot-time
- RAM is continuously allocated/removed from free list for kernel and user processes
- RAM is eventually exhausted and therefore reclaimed from users and few select kernel locations and freed.
- kswapd - Linux page reclaim kernel thread.

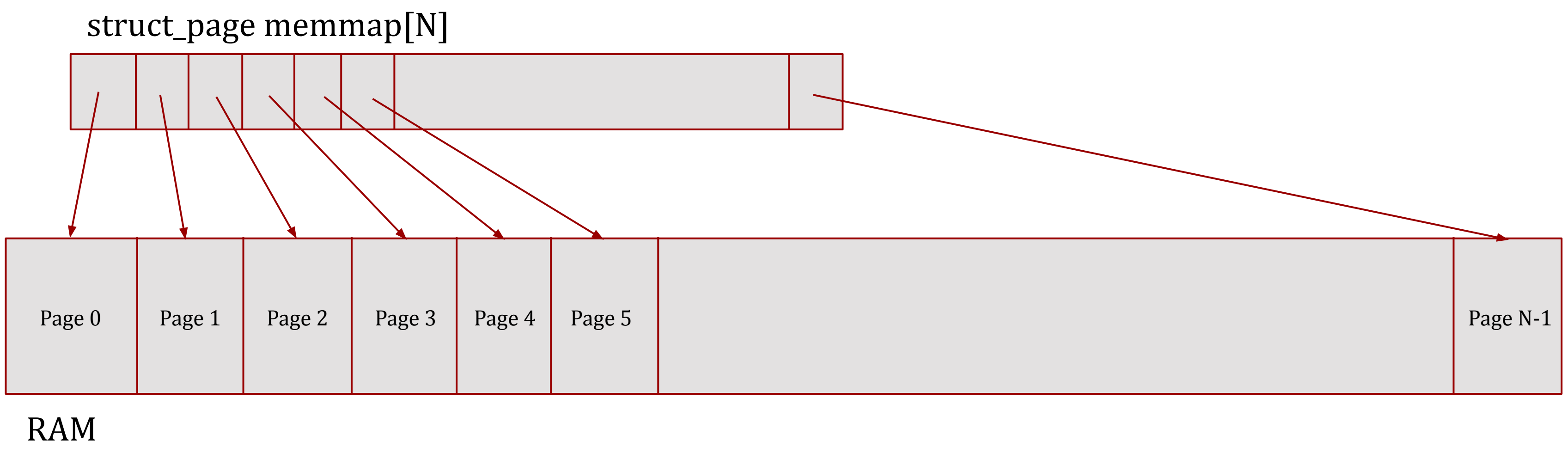
# We talked about reclaiming...

- How do we find out where page mapped?
- How do we keep track of paged out anonymous memory?

Lets discuss a few of the core MM data structures

# Per-page frame information

- struct_page to track each page of RAM
- mem_map is an array of struct_pages
- The Nth page of RAM is tracked by mem_map[N]
- Each struct_page is 64 bytes and describes one 4096 byte page of RAM
  - The mem_map consumes 2^6/2^12 or 1/64 of all the RAM(64/4096)
  - This is HUGE(hence the desire for larger page sizes)!!!



# The ‘struct page’ fields

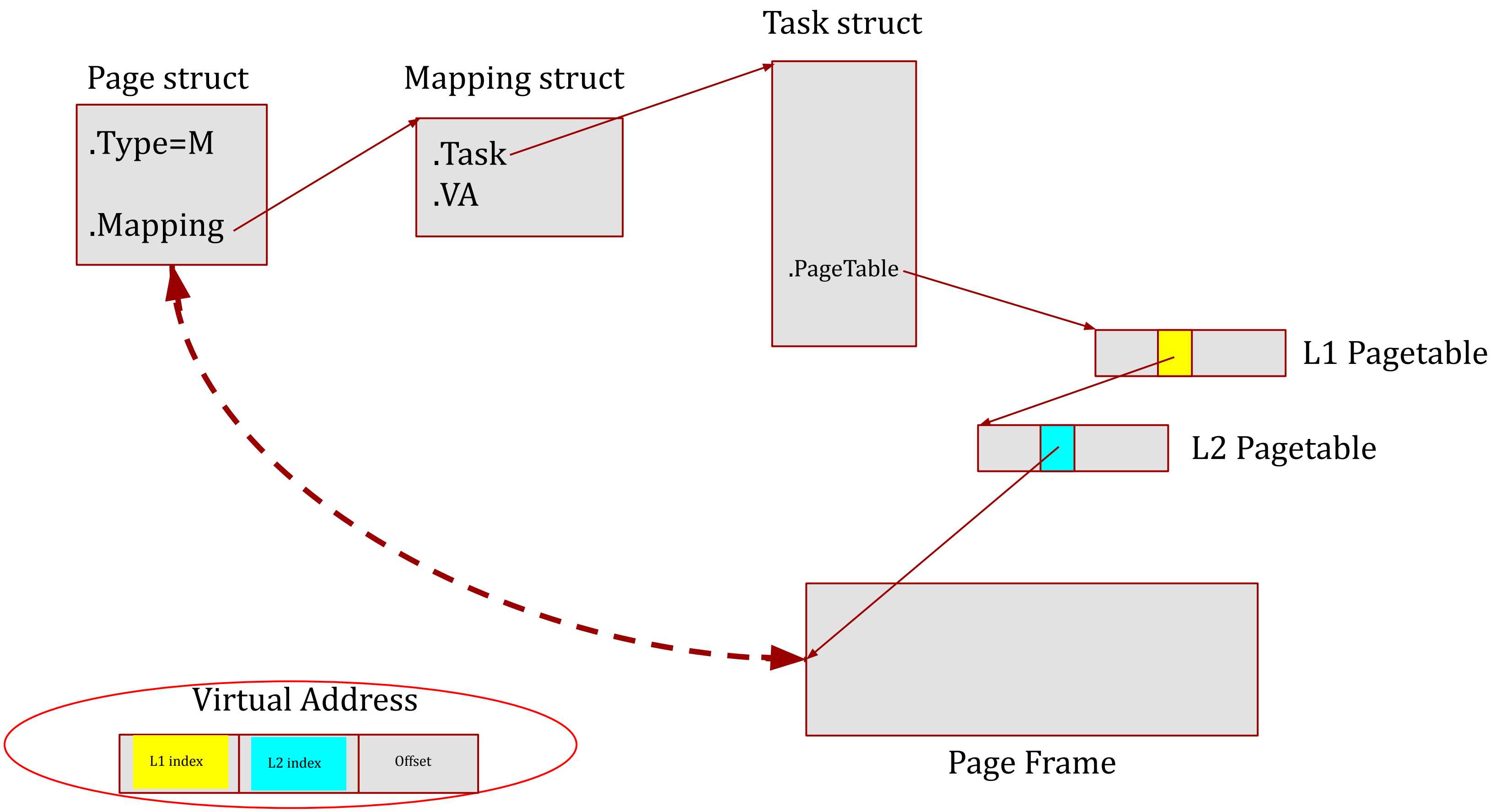
struct page ( 8 longwords )

| | | | |
|---|---|---|---|
| **flags** | _count | _mapcount | private |
| mapping | index | lru.next | lru.prev |

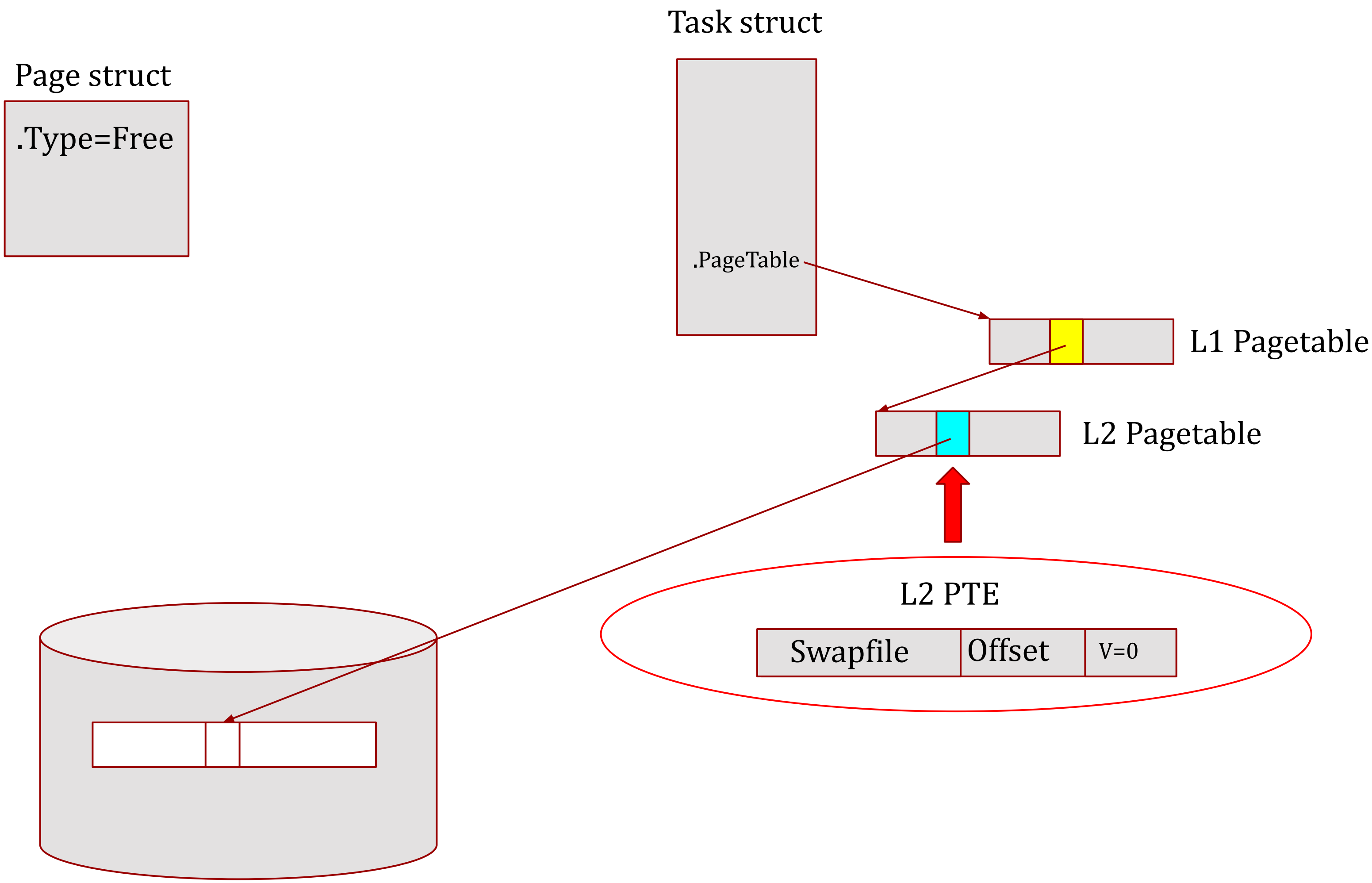
The ‘struct page’ definition is in the <linux/mm.h> header

along with the declaration for the ‘mem_map[ ]’ array

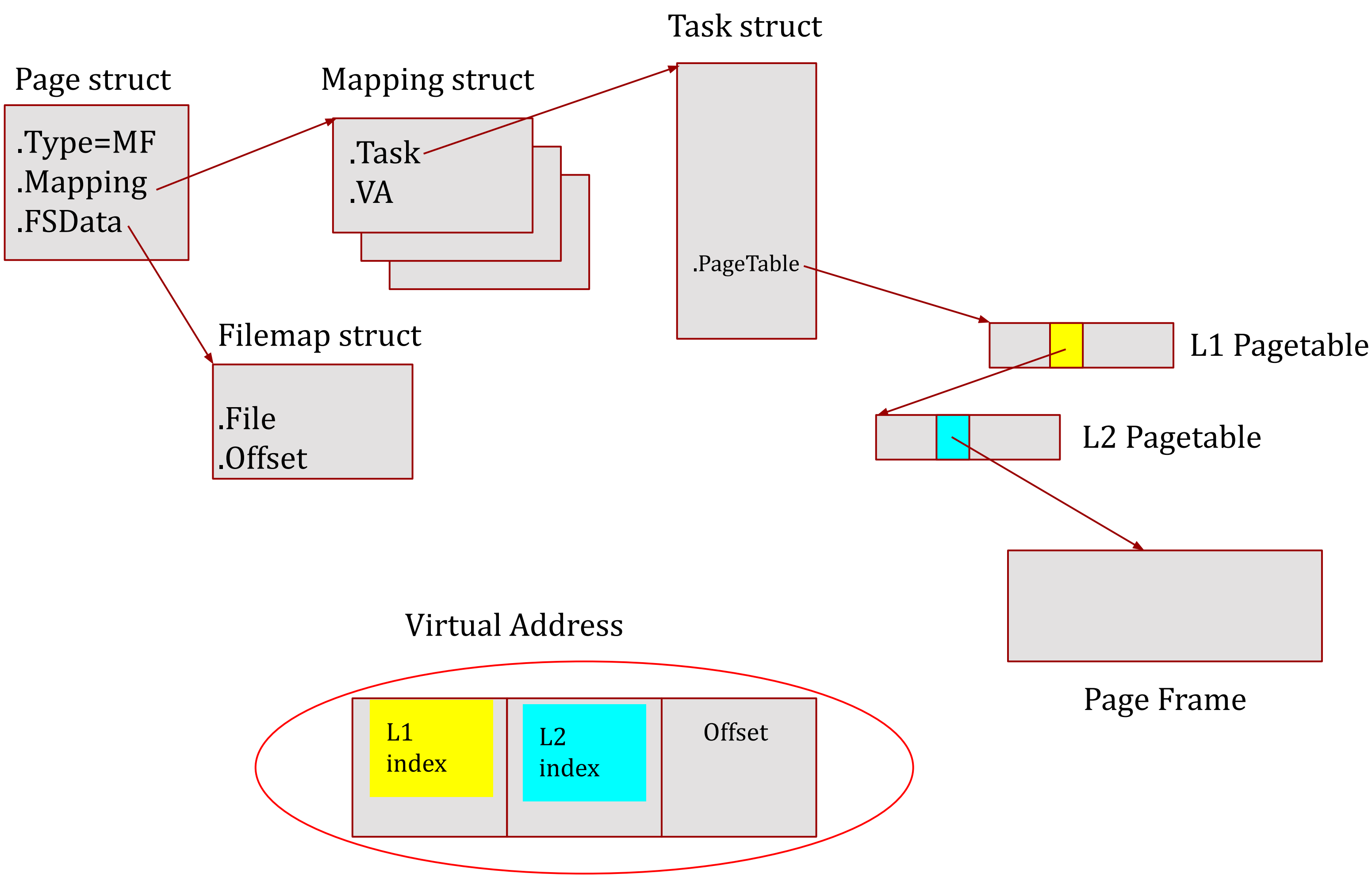
# Anonymous Mapping Data Structures



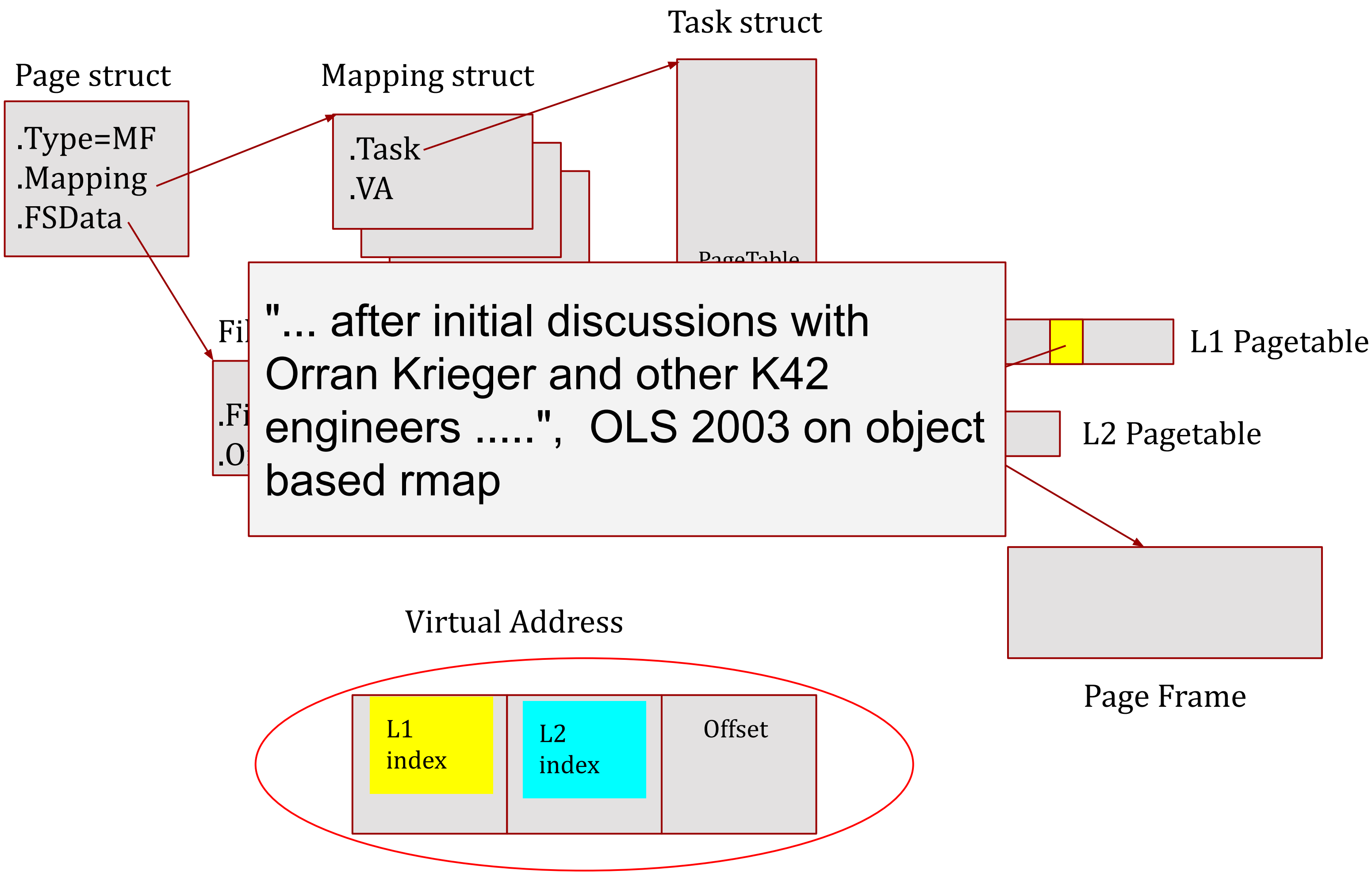
# Anonymous page swapped out



# Mapped File Data Structures



# Mapped File Data Structures



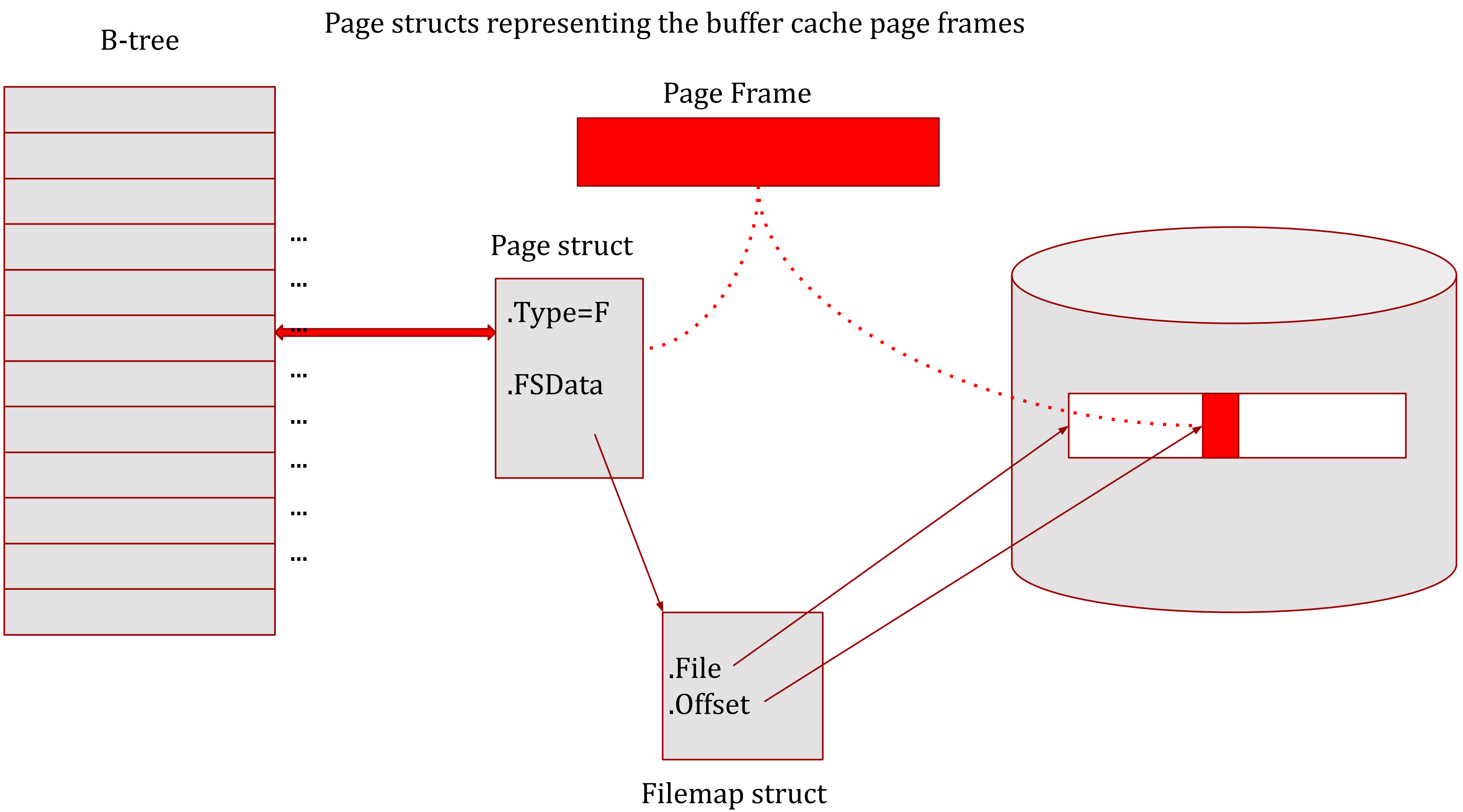

"... after initial discussions with Orran Krieger and other K42 engineers .....", OLS 2003 on object based rmap

# The Buffer Cache

# The Linux Page Cache

- Page cache is the Linux term for Buffer Cache
- Works just like we talked about...
  - Uses physical memory/Page Frames to cache file system data.
  - Improves file system performance by orders of magnitude by eliminating **most** file system IO and associated blocking.
- Integrates memory management with file systems.
- The page structures representing page frames containing the filesystem data are inserted into the hash based on File system Object/Offset tuples

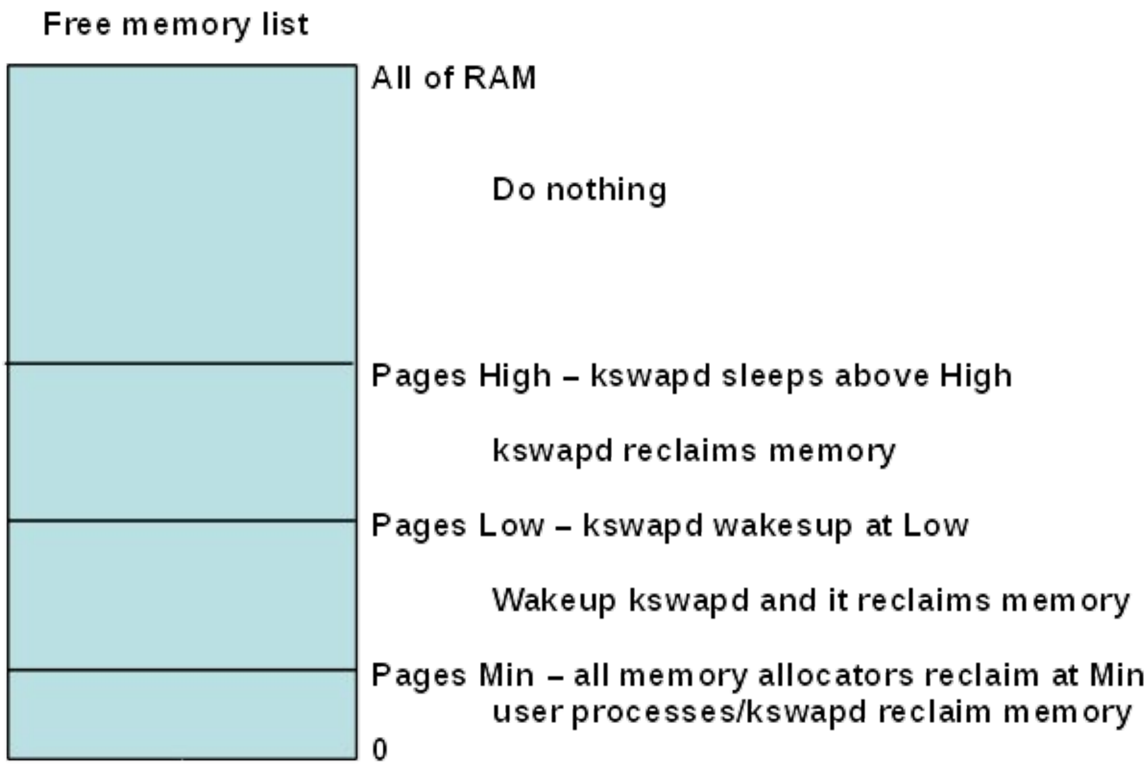
# The Linux Page Cache



# Reclaiming memory

# When do we reclaim?

## Memory reclaim Watermarks



# Reclamation issues

- Much cheaper to reclaim memory that is file backed.
- Much cheaper to reclaim unmapped pages
- We discussed a whole bunch of policies; complexity of data structures is a huge deal
  - VMS & Windows uses local working sets
  - Linux uses simple global second chance like approximation of LRU

# Linux Memory Reclaiming

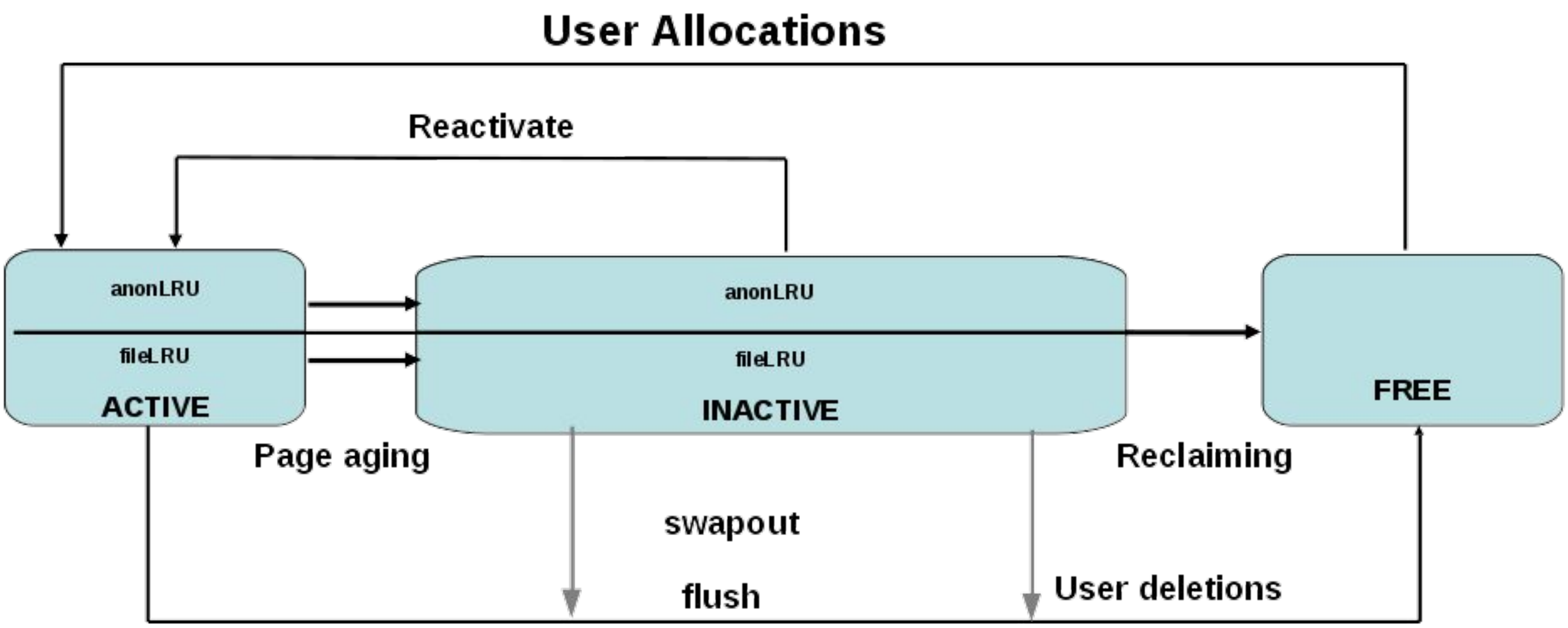
- Two sets of LRU lists (pagecache & anonymous):
  - active_list - contains about 2/3 memory
    - newly allocated pages inserted on tail
  - inactive_list - reclaim candidates
- Uses second chance algorithm:
  - move referenced pages to tail of active_list
  - kswapd clears reference bits from bunch of pages at head of active list, moves to tail of inactive
  - if page gets to head of inactive list,
    - if reference bit set, moved to tail of active list
    - else used

# Reclaiming Memory

- The percent of memory reclaimed from anonymous/pagecache is proportional to the memory they consume:
- Typically > 90% of memory is unmapped pagecache
  - kernel keeps pagecache clean
    - reclaim is very simple/fast
  - dirty pages must be written before reclaim.
    - reclaim more complex/slow
- Typically less than 10% of memory is Anonymous.
- All mapped memory is unmapped before reclaim.
  - PTEs must be located(reverse mapping structs)
  - PTE valid/present bit is cleared
  - dirty Anonymous pages are written to swap
    - swap device/location stored in PTE.
  - dirty file mapped pages are written to file.

# Linux Memory Reclaiming

## Per Node / Zone split LRU Paging Dynamics



# Linux Slab Cache

- Customizable per-subsystem memory object allocator.
- Layered on top of the Linux buddy page allocator.
  - Allocates pages of memory mapped into kmalloc space.
  - from 1 page(4KB) up to 1024 pages(4MB).
- Any subsystem can create its own object cache:
  - kmem_cache_create()/kmem_cache_destroy()
  - kmem_cache_alloc()/kmem_cache_free()
- An option on creation is whether the objects are reclaimable.
- File systems use reclaimable slab cache for in-memory copies if on-disk data structures:
  - inode(inode_cache)
  - directory entries(dentry_cache)
- All other kernel data structures are not reclaimable:
  - task, mm, vma, etc.

# So we have

- Page table structure
- Allocate and de-fragment
- How do we find out where a page is mapped
- Buffer cache
- How we reclaim memory

Now we can talk about how a page fault actually works

# Handling page faults

# Page Fault details

1. User code touches VA with PTE.valid not set.
2. HW traps into kernel page fault handler
   a. CPU enters kernel mode
   b. switches to the per-thread kernel stack
3. PF handler verifies VA using MM structures
4. PF handler calls User VA specific handler:
   a. Anonymous fault handler
   b. Mapped-File fault handler
5. PF handler switches to user stack & REI to faulting user instruction.

# Anonymous Page Faults

- Initial anonymous page faults are Zero Filled on Demand(ZFOD)
- subsequent anonymous faults are:
  - swapped-in(if they were reclaimed)
  - Copy-On-Write(if the page is not shared)
  - Linux supports shared anonymous memory so parent/child tasks can share memory regions.

# Mapped File Page Faults

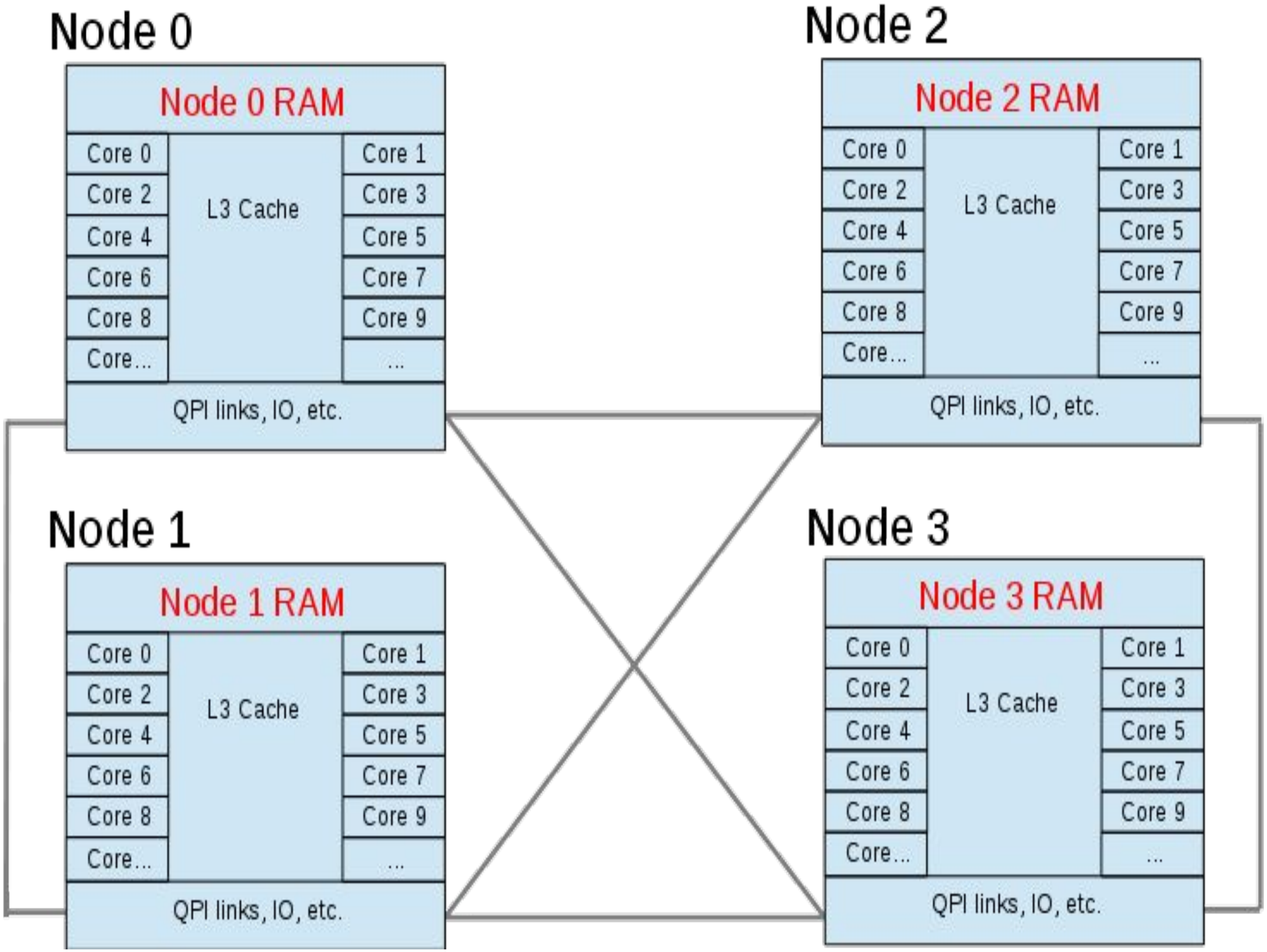
- As we said earlier, all file data is cached in memory page frames via the Buffer Cache.
- Mapped file page faults simply map Buffer Cache page frames into the faulting VA via the process's PTEs.
  a. locate page in the Buffer Cache
     i. (reading it into the pagecache on a miss).
  b. map Buffer Cache page frame via the PTE.

# COW Faults

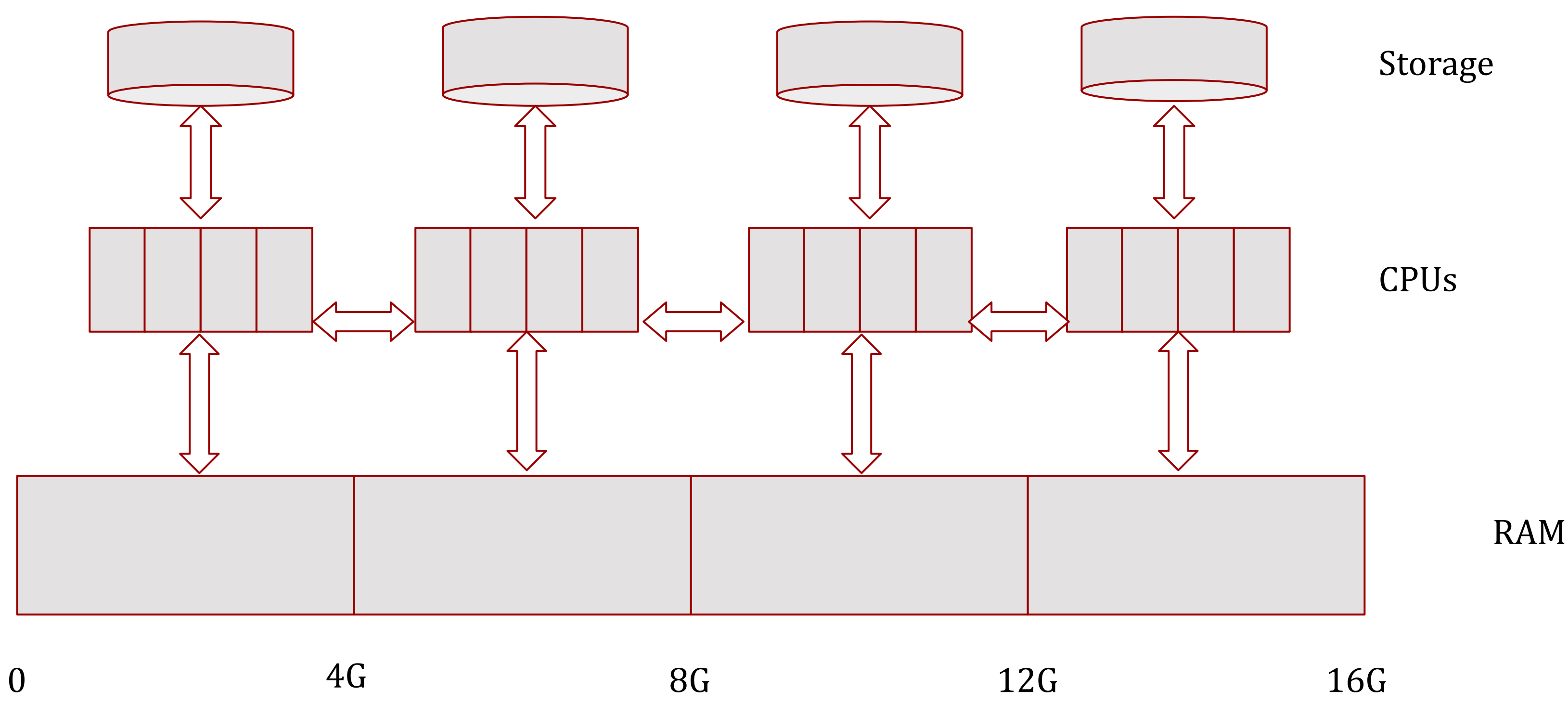
- Copy On Write fault gives private copy of page
- Fork() duplicates page tables in parent/child.
  - All load/read instructions use shared pages.
  - store/write instruction creates private page.
- Files mapped “shared” always share page.
  - This is “initialized data”.
  - writes/stores get written back to disk
- Files mapped “private” use COW.
  - This is “uninitialized data” or BSS.
  - All load/read instructions use shared pages.
  - store/write instruction creates private page.

# One last thing...
# NUMA

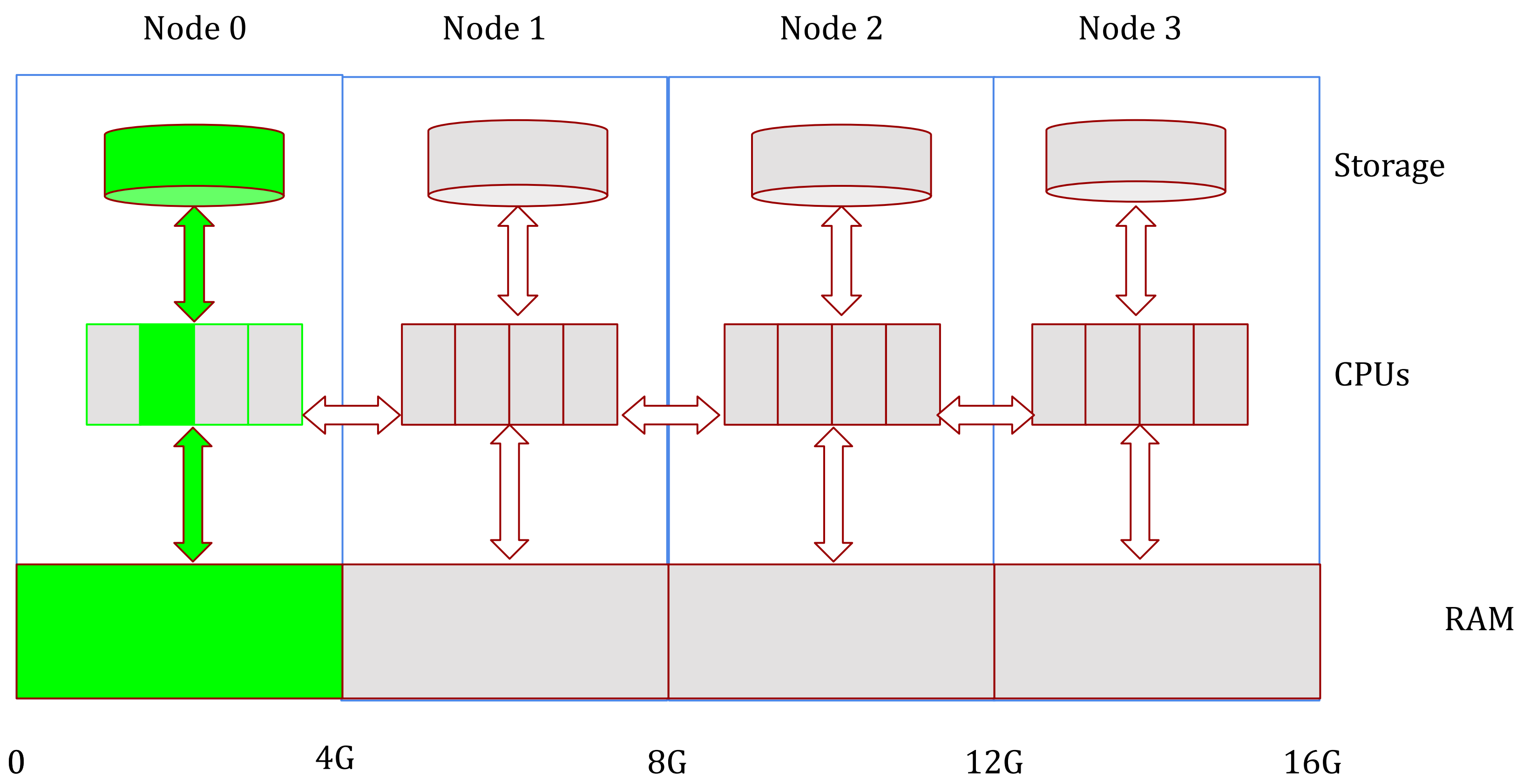
# Typical Four-Node NUMA System



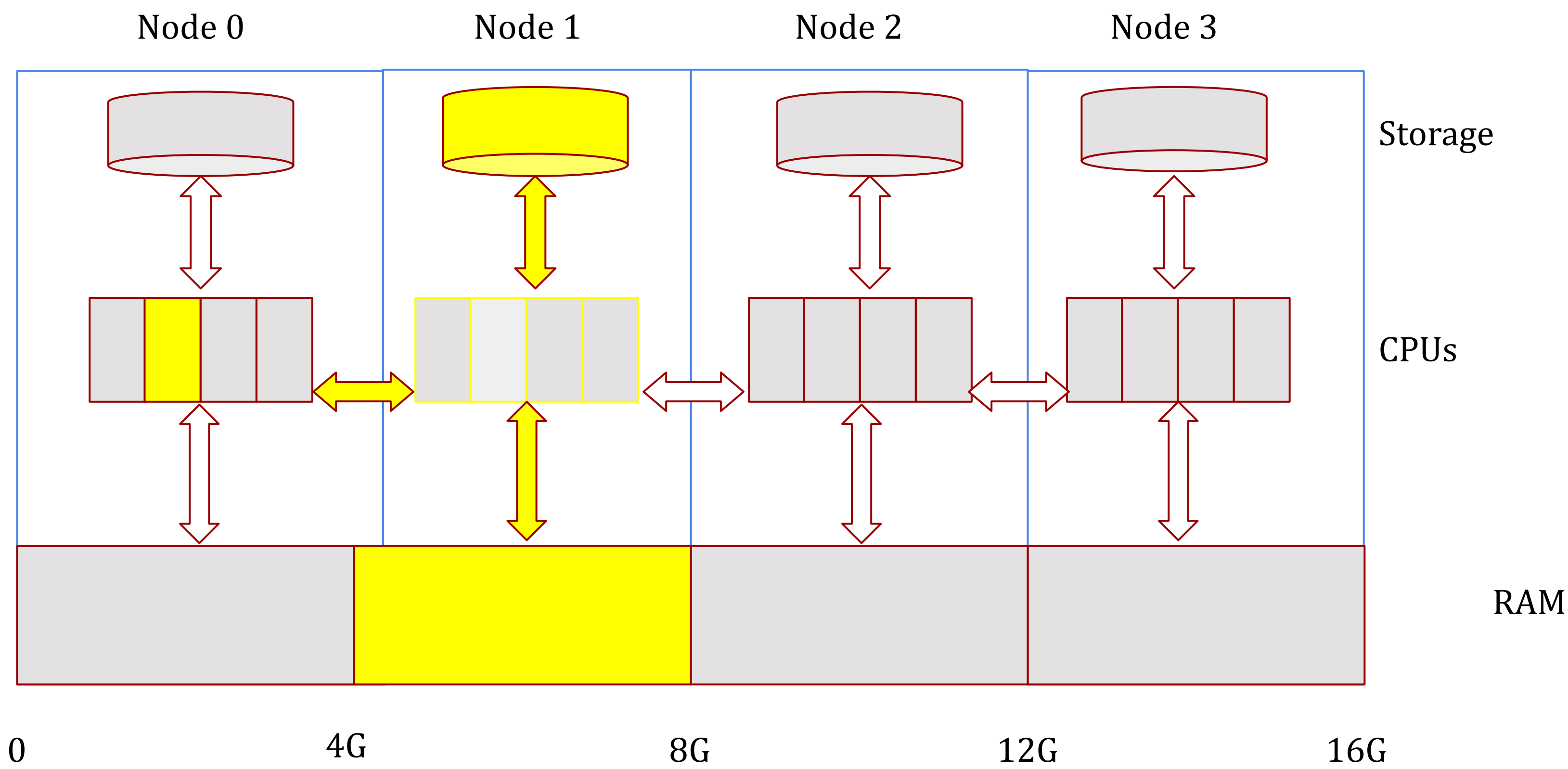
# NUMA



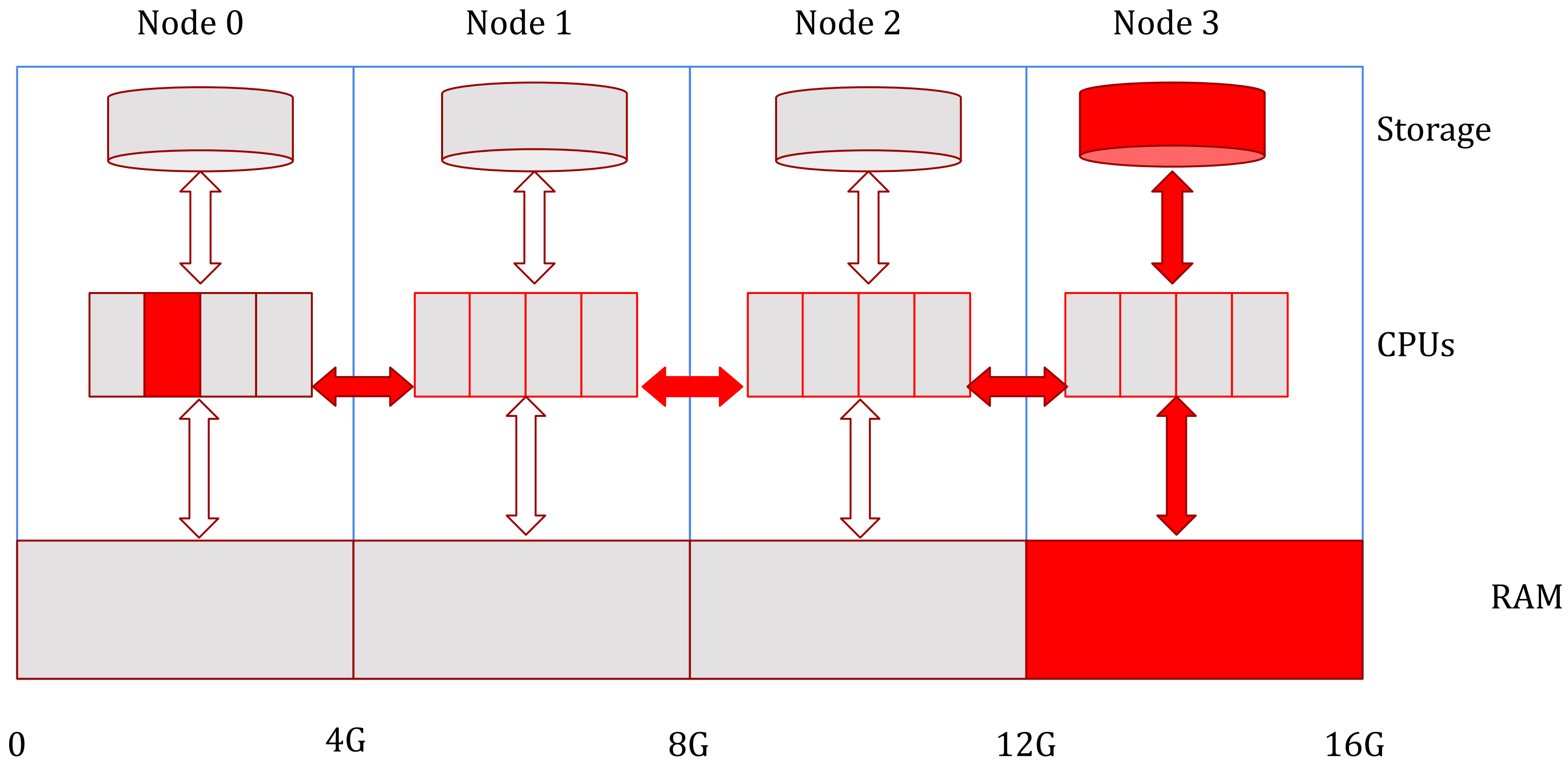
# NUMA local memory/storage access
# Best possible case



# NUMA remote memory/storage access
# Best case



# NUMA local memory/storage access worst case



# Looking at NUMA node details

```
[root@localhost ~]# numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5
node 0 size: 2047 MB
node 0 free: 1841 MB
node 1 cpus: 6 7 8 9 10 11
node 1 size: 2047 MB
node 1 free: 1947 MB
node 2 cpus: 12 13 14 15 16 17
node 2 size: 2048 MB
node 2 free: 1938 MB
node 3 cpus: 18 19 20 21 22 23
node 3 size: 2048 MB
node 3 free: 1922 MB
node distances:
node   0   1   2   3
  0:  10  20  20  20
  1:  20  10  20  20
  2:  20  20  10  20
  3:  20  20  20  10
```

4 vNUMA nodes

# Linux NUMA support

- NUMA aware
  - Memory Allocator - memory allocated on node where processes are running
  - Scheduler - runs processes where memory is allocated
  - I/O system - uses I/O device local to the node where processes are running
- NUMA Balancing - kernel attempts to dynamically migrate processes where memory is allocated.
- Kernel allocates data structures and page tables on node where processes are running.