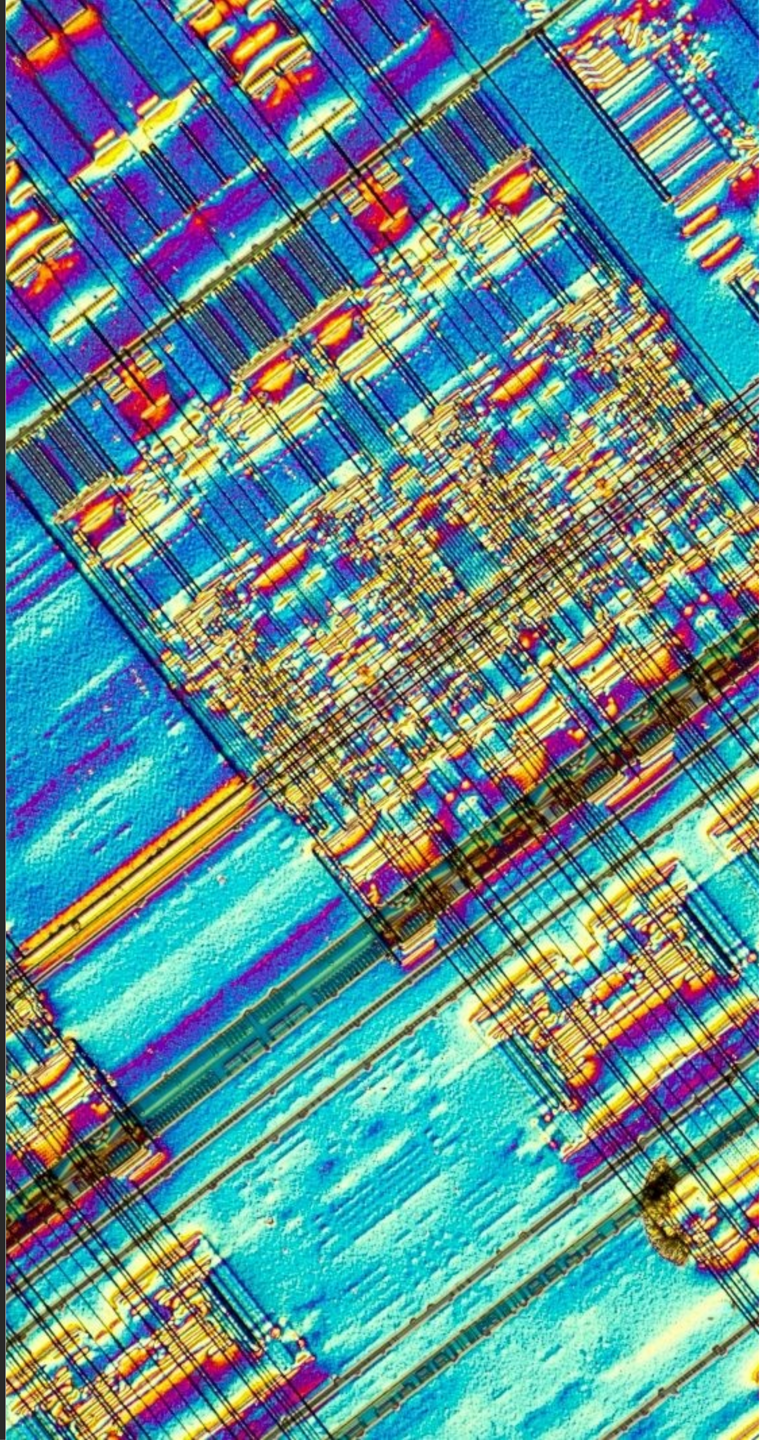
# Memory Management Subsystem

# Overview

- Introduction to virtual memory, paging, page tables and address space
- Linux memory layout
- Memory slabs
- Memory allocation and management
- Memory API
- GFP flags



# Early days of memory management

# No need for memory management

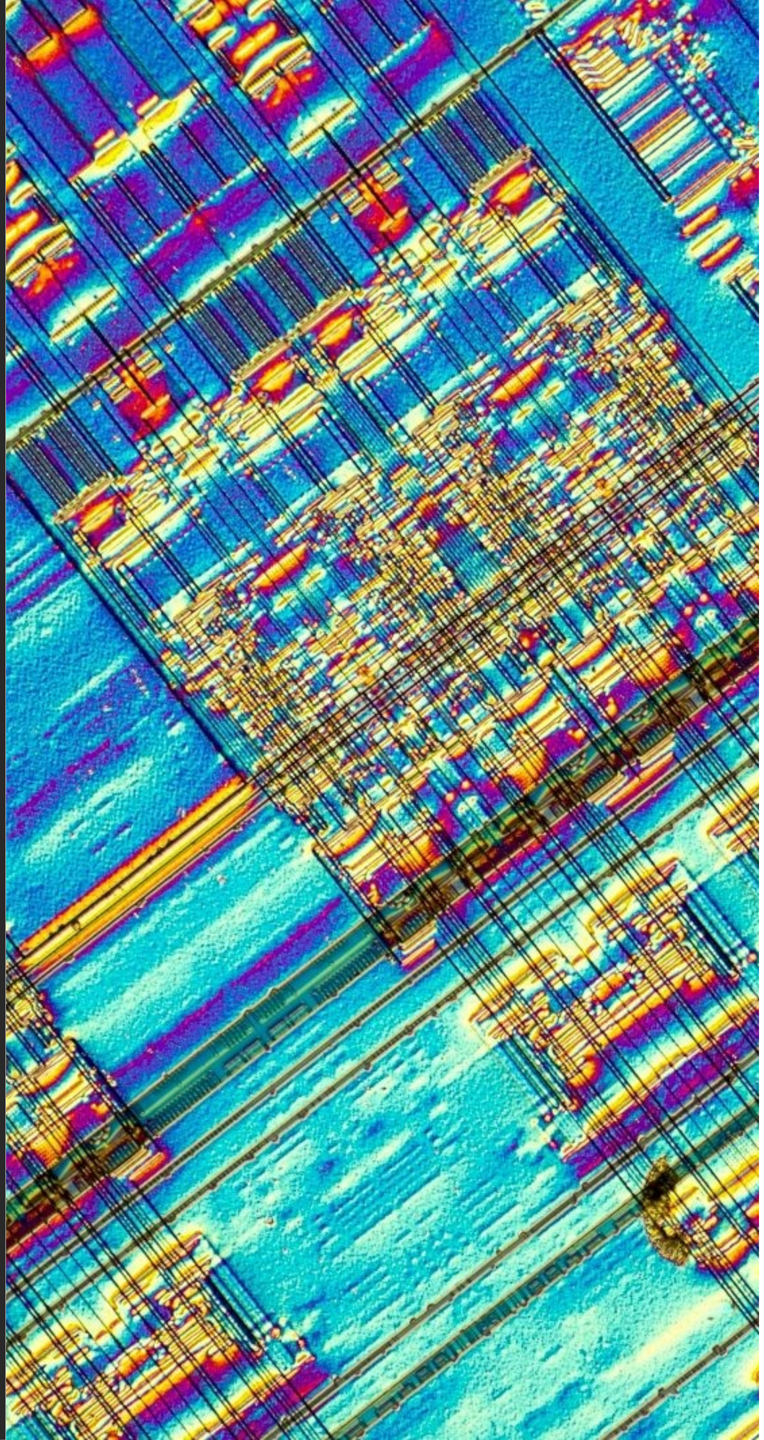
- No multi-user or multi-programming computers
  - We could only have a single program running at a time
- Sometimes no OS was used
- Early days OS'es were just a small collection of libraries for common hardware access

# Better computers = New problems

- New computers brought more resources
  - faster CPUs
  - bigger amounts of memory
- Running a single program at a time became a waste of power, so we reached a new era.
  - Multi-programs
  - Multi-user
  - Multi-problems

# New era problems

- How to load many programs  into memory at the same time and:
  - Programs don’t need to be loaded on different addresses
  - They can’t access each other memory areas
  - Programs can’t monopolize the whole physical memory, starving other programs.
- Virtual memory comes for the rescue



# Virtual memory

# Virtual memory concepts

- Memory management technique where the OS (with hardware support) enables the system memory to be shared between programs
  - Simplify the memory addressing for processes
  - Allow full isolation of memory between running programs
  - Memory allocated on-demand

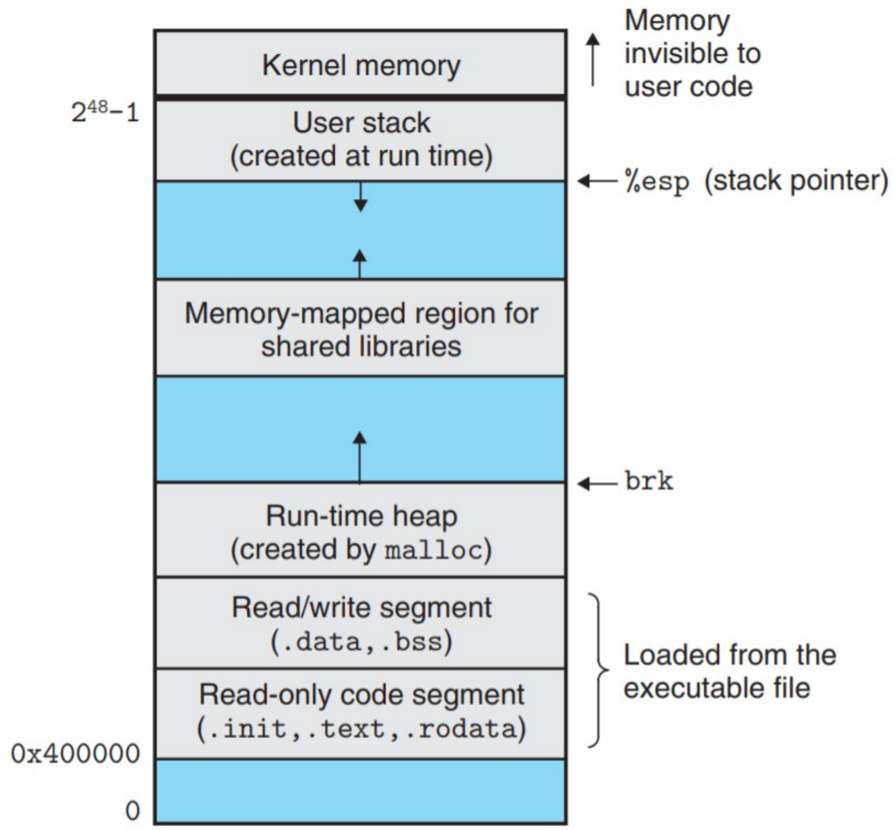


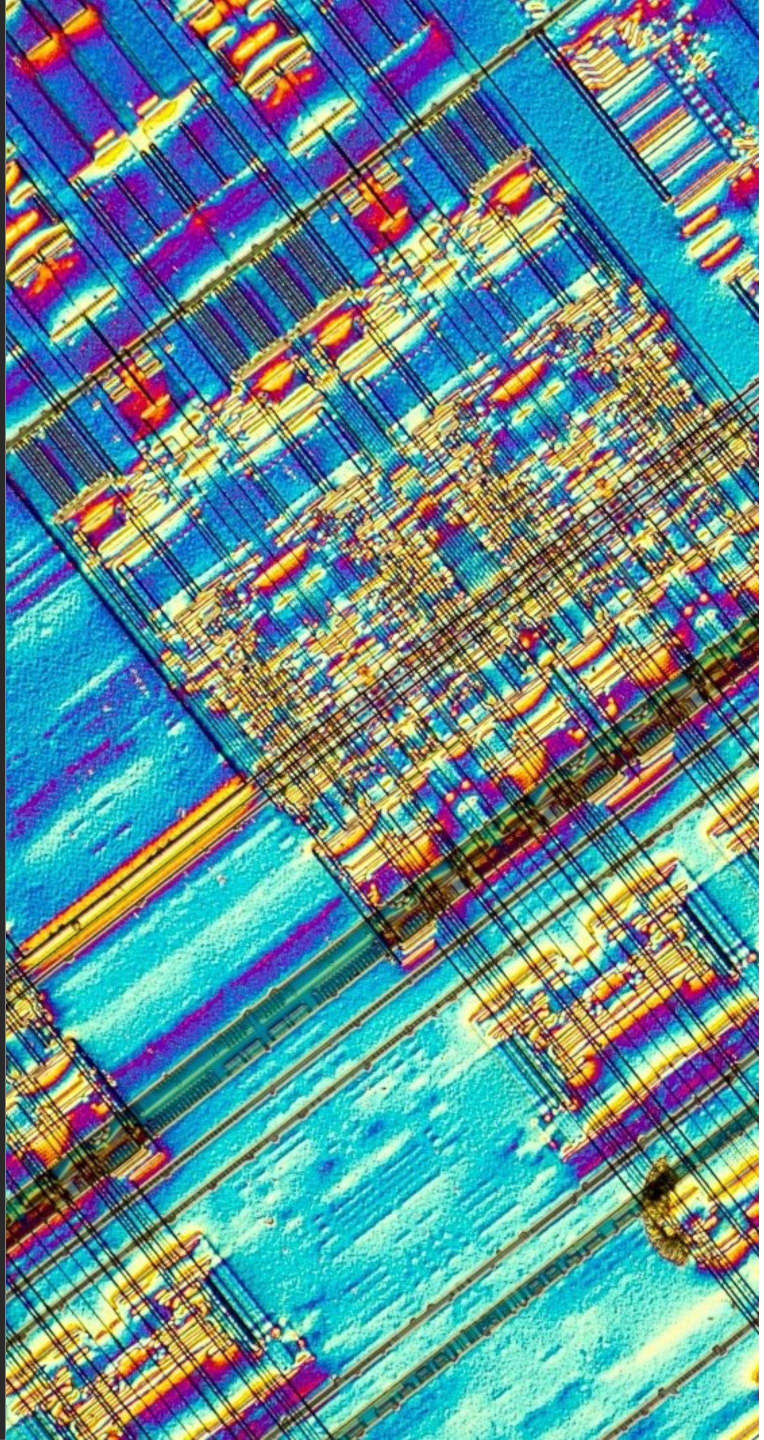
Red Hat

# New abstractions

- Transparency and illusion - it literally fools programs
  - An individual **Address Space** for each program.
- And this is how a program “sees” memory…

# Address Space





# Memory addressing and address translation

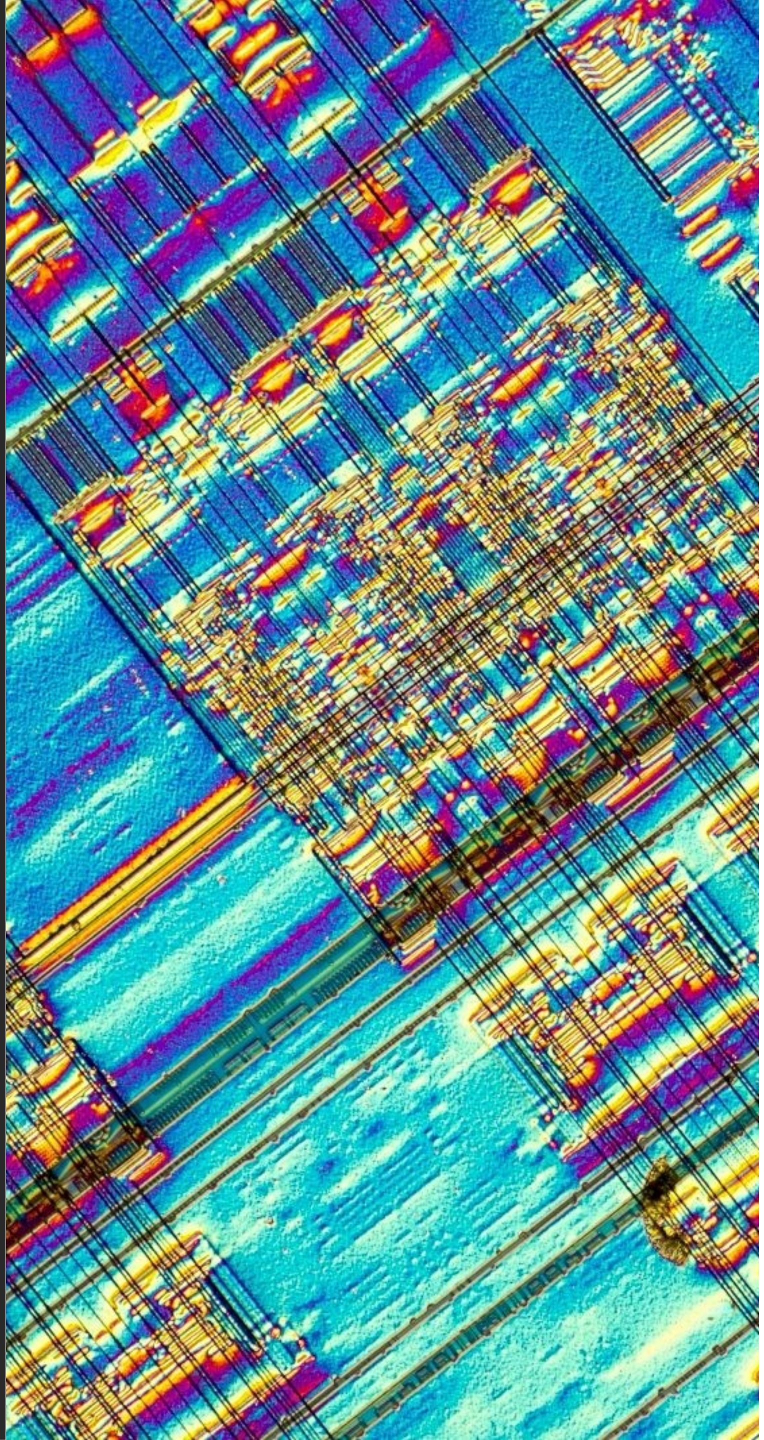
Red Hat

# The three memory addresses

- A “memory address” may have different meanings:
  - The Logical address
    - Generated using memory segments
  - The linear address
    - The virtual address
  - The Physical address
    - Address of memory cells in chips

# Address translation

- Addresses generated by programs are **virtual addresses**
- Physical <-> memory translation
  - Hardware’s low-level circuitry make the translations more efficient.
  - every fetch/load/store causes an address translation
  - OS is responsible for managing it (control free/used memory, access, etc)
  - MMU transforms physical into linear addresses

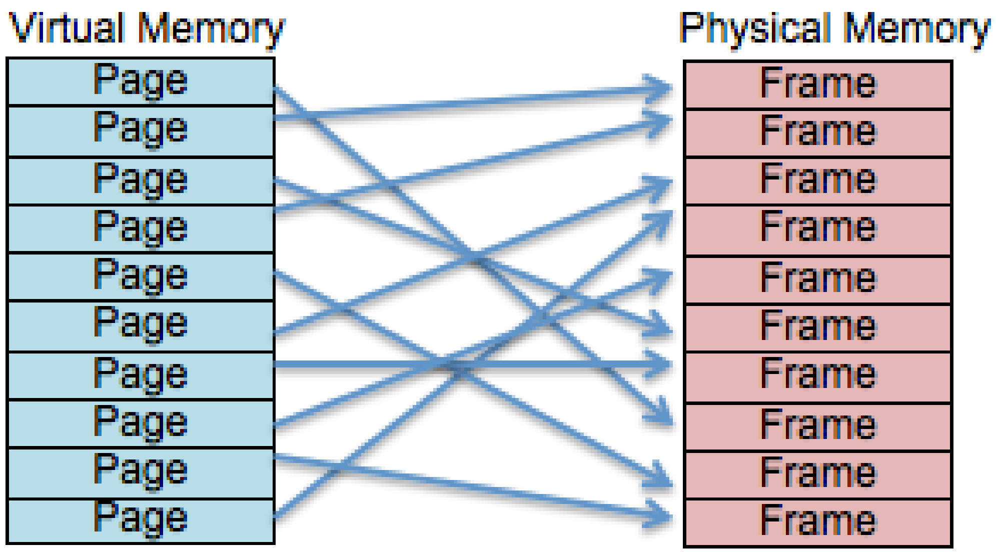


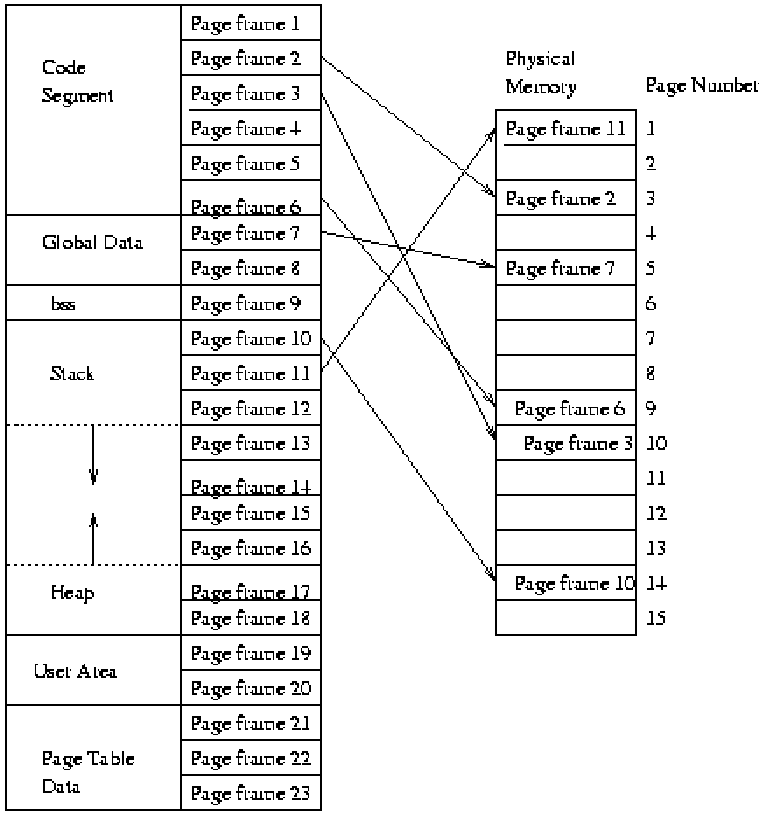
# Paging

# Memory paging

- Physical memory is split into fixed-sized “slots” named:
  - **Page Frames**
- Processes address space are now divided in pages and not in segments
- A page IS NOT a page frame
  - Page = Chunk of data
  - Page frame = Physical “slot” within the machine’s memory

# Page vs Frame



# Memory paging #2

- Pages are easier to manage
- Results in less fragmentation
- Memory usage is tracked through a “Page Table”
- Entries in the page table are called **Page Table Entry** (or PTE)

# Page Tables

- Indexes all the pages used in the system
- Stores and indexes several PTEs
- Each PTE contains the needed information to perform an address translation Physical <-> Virtual
- Page tables are “per process” data structures
  - Paging is slow - TLB for the rescue
  - Different architectures and OSes implement it in different ways

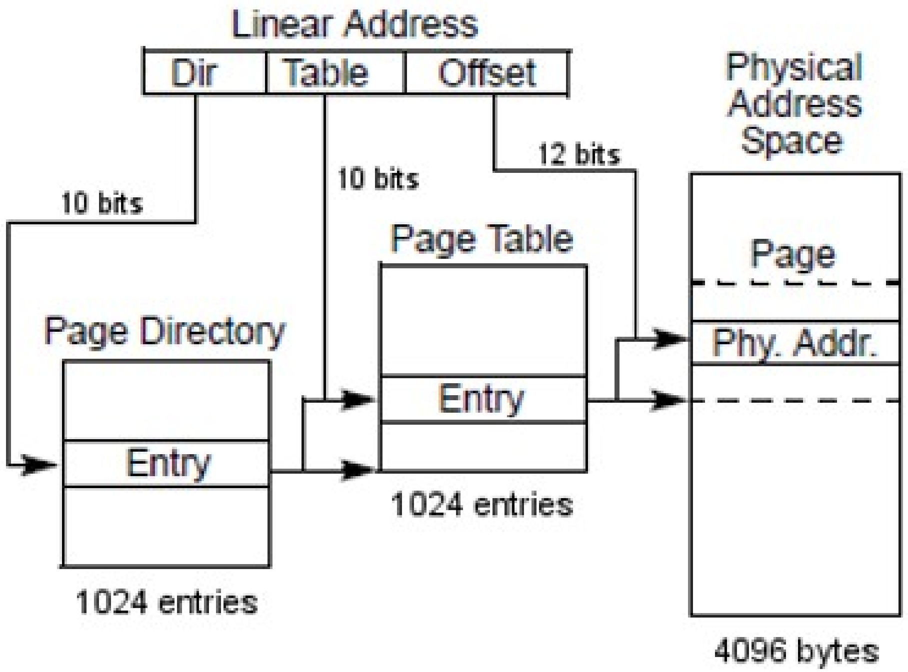
# Page Tables implementation

- We could implement a simple page table in a Linear way (using x86 32-bit as example), where given an address:
  - Bits: 12-31 -> describe the page index
  - Bits: 0 11 -> Offset within the page
- This is really simple, but has a big issue:
  - Having PTEs of 4 bytes each, would required 4MiB ram for each process in the system
  - This is too much memory just for memory management

# Page Tables implementation #2

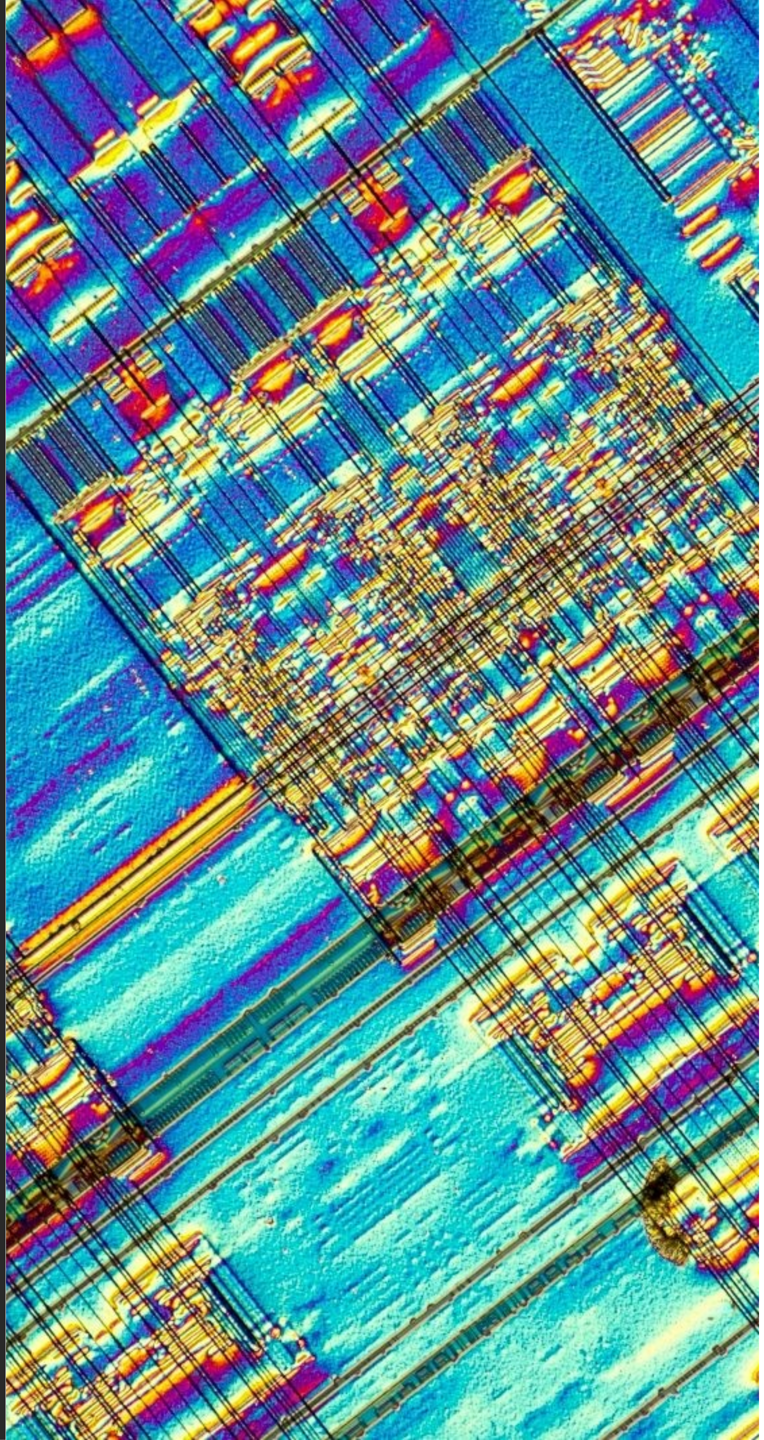
- Preventing excessive memory consumption can be reached by
  - Employing a multi-level page table
- On 32-bit systems, the linear address space is split into 3 levels:
  - Page Directory (10 MSB)
  - Page table Entry (next 10 bits)
  - Offset (the last 12 LSB)

# Address translation using the page table



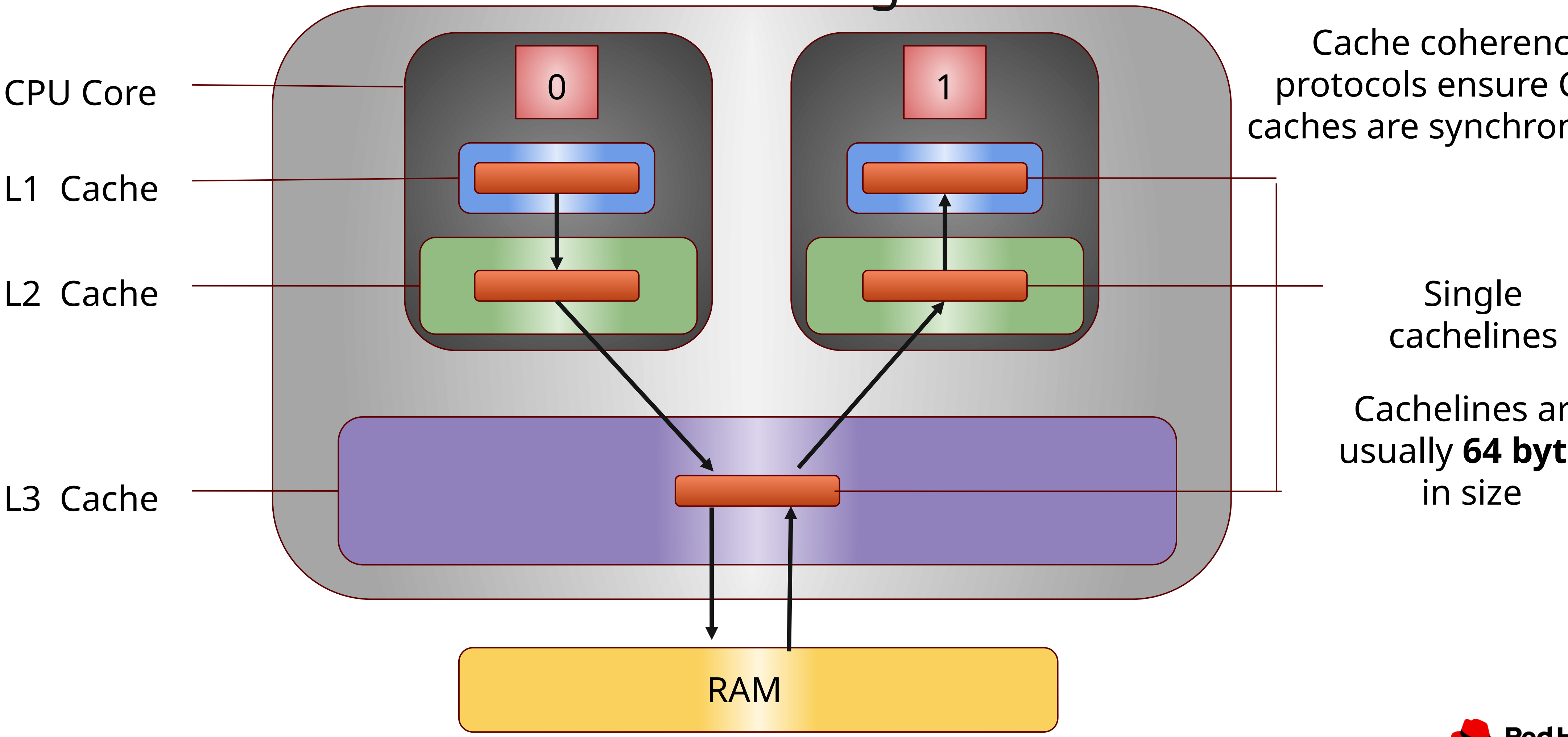
# Multi-level paging details

- Not all pages within a virtual address space need to be mapped to a physical page
  - Processes usually don’t have the whole address space allocated
- An attempt to access a not yet mapped virtual address, will cause the CPU to raise a **“Page Fault”** exception, passing the control back to the operating system.
  - The OS will then map that page table
- The MMU does play a big role here, but we won’t dive into hardware details



# CPU caches Reducing thrashing , and false sharing

# Basics of CPU caching workflow

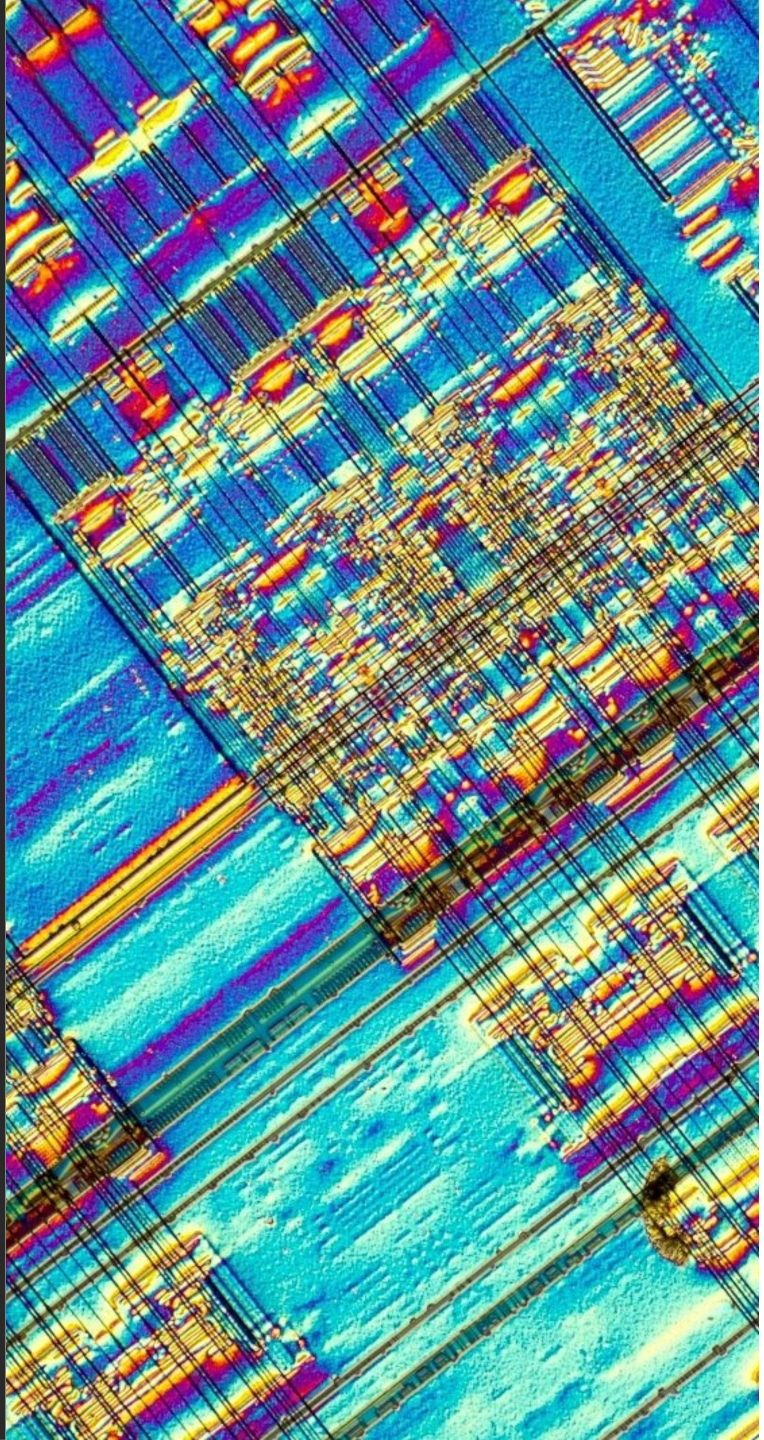



Cache coherence protocols ensure CPU caches are synchronized.

Cachelines are usually **64 bytes** in size



Red Hat



# Linux kernel’s memory management

# Handling memory within kernel

- Memory allocation within a kernel is a different beast when compared with user-space.
  - Allocating memory isn't always easy, specially on embedded systems where memory is short
  - Kernel often can't sleep.
- We shall see how it works

# Linux Paging

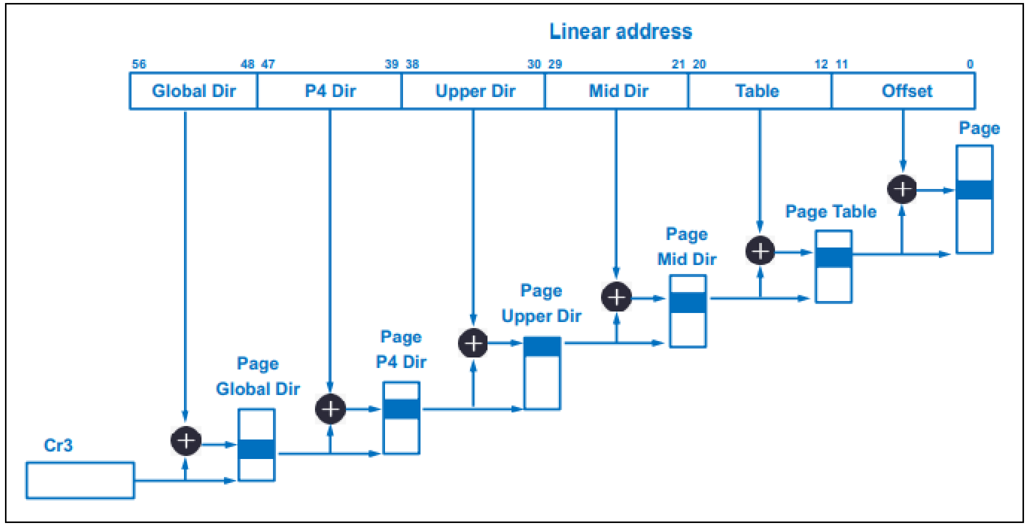
- Quick recap
  - Memory is handled by the machine and kernel itself using MMU when available to maintain the page tables and handle address translation
  - Page size is architecture dependent
- Every **physical page** is represented by a **page** data structure
  - **struct page** goal is to describe the physical memory not the data within it.

# Linux Page Tables

- Linux defines page tables as a hierarchy (multi-level page tables)
- The code for the specific architectures will map this hierarchy to the hardware restrictions.
- The number of levels in the page table varies depending on the architecture
- Top-level address is stored in a CPU register

# Linux Page Table diagram

- PGD -> P4D -> PUD -> PMD -> PTE
  - P4D was introduced to handle 5-level tables, only used with 5 levels, otherwise, it's folded



# Process Address Space

# Process address space

- Memory region used by each process
- It can (and usually is) way larger than available physical memory
- Consists of:
  - Virtual memory addressable by a process
  - Addresses within the virtual memory the process is allowed to use

# Process address space #2

- Flat address space given to a process
- Architecture dependent
- Processes see the same addresses, but the address space is unique for each process
- Address spaces can be shared among process (Threads)
- The process does no have access to all addresses within the address space

# Process address space #3

- Address spaces are split into memory areas that can be dynamically added/removed (With kernel's help)
- Memory areas have their own associated permissions (R, W, X)
- Don't respect the permissions and you get a Segmentation fault

# address space descriptor (aka Memory descriptor)

- mm_struct - represents a process's address space
- Linked to the process's task_struct via current->mm field



Red Hat

# Kernel threads address space

- Kernel Thread definition:
  - A process without user context
- kernel threads have no process address space
  - No associated memory descriptor ->mm field is NULL
- No userspace pages, so, no page tables.
- So, without page tables, without a memory descriptor…
  - How kthreads deal with memory then?

# Kernel threads address space #2

- They “borrow” the memory descriptor of whatever task ran before it.
- A process is scheduled…
  - The address space referenced by the ->mm field is loaded
  - The active_mm field is updated to this new address space

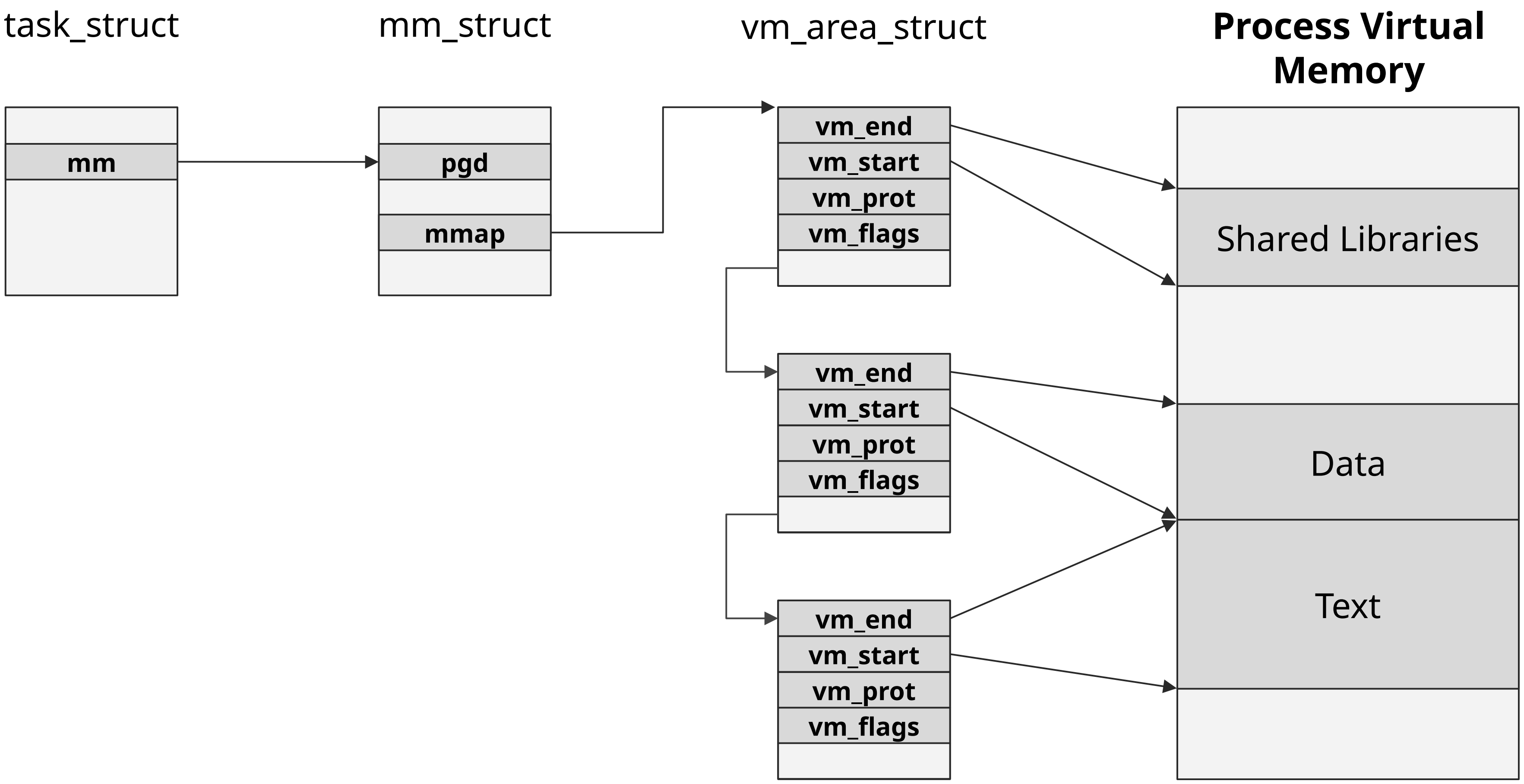
# Kernel threads address space #3

- A kthread is scheduled…
  - The kernel sees the NULL ->mm field, and keeps the previous address space still loaded.
  - The ->active_mm field of the kthread's process descriptor is updated to refer to the same address space of the previous process (currently loaded).
- The kthread can use the previous process page tables as needed.
- Kthreads never access userspace pages AND all address space information related to kernel memory, is the same for all processes.

# Virtual Memory Areas (VMAs)

- vm_area_struct descriptor
- Represent individual memory areas within the Address Space
- Each memory area has its own properties
  - Permissions, associated operations...
- Each VM can represent different types of memory areas
  - mmapped files, user-space stack...

# Virtual memory areas #2



# Virtual memory areas (aka VMAs) #3

- VMAs are unique for the associated mm_struct
  - Each process has its own individual address space
  - We could have two processes mapping the same file in their address spaces, and yet, each one will have an unique vm_area_struct for that file map
- Threads sharing the same address space will also share the same VMA regions.

# Virtual memory areas (aka VMAs) #4

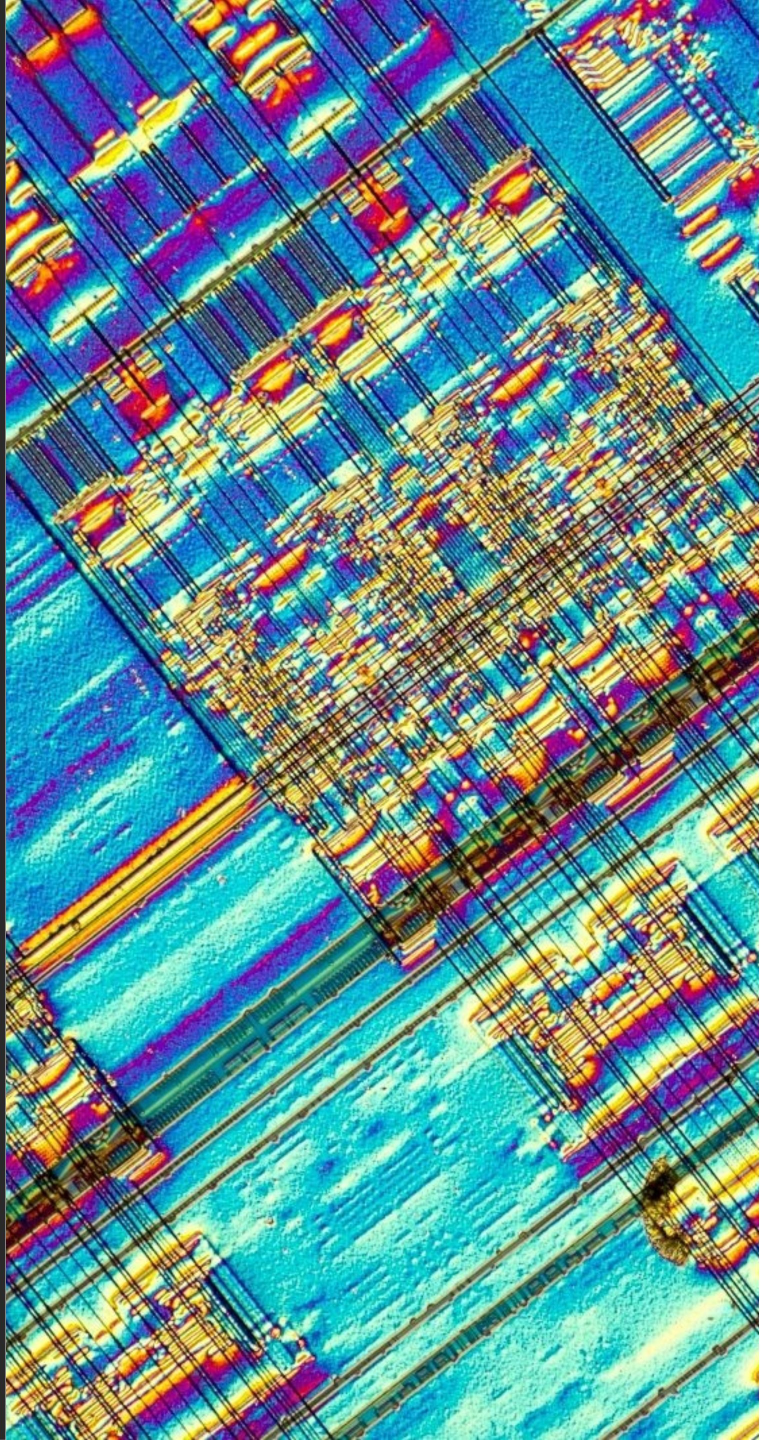
- Each VMA have its own permissions and purpose
- vm_page_prot and vm_flags configure such permissions
- Some of these settings are directly influenced by system calls such as madvise()

# VMA Operations

- Similar as filesystems behavior depends on the internal filesystem implementation
- VMAs operations also can be customized depending on what is mapped on such memory region
- Filesystems set specific vma operations to deal with mmapped files, so the kernel know what to do in situations such as
  - Page faults, page mapping, write specific page frames
- Not mandatory, and the VFS provide some generic functions

# VMA Allocation

- New VMAs are allocated through do_mmap()
  - This is not (totally) related to mmap() syscall
- Possibly, it can simply merge the new request into an existing area



# Memory zones

# Linux divide memory in different zones

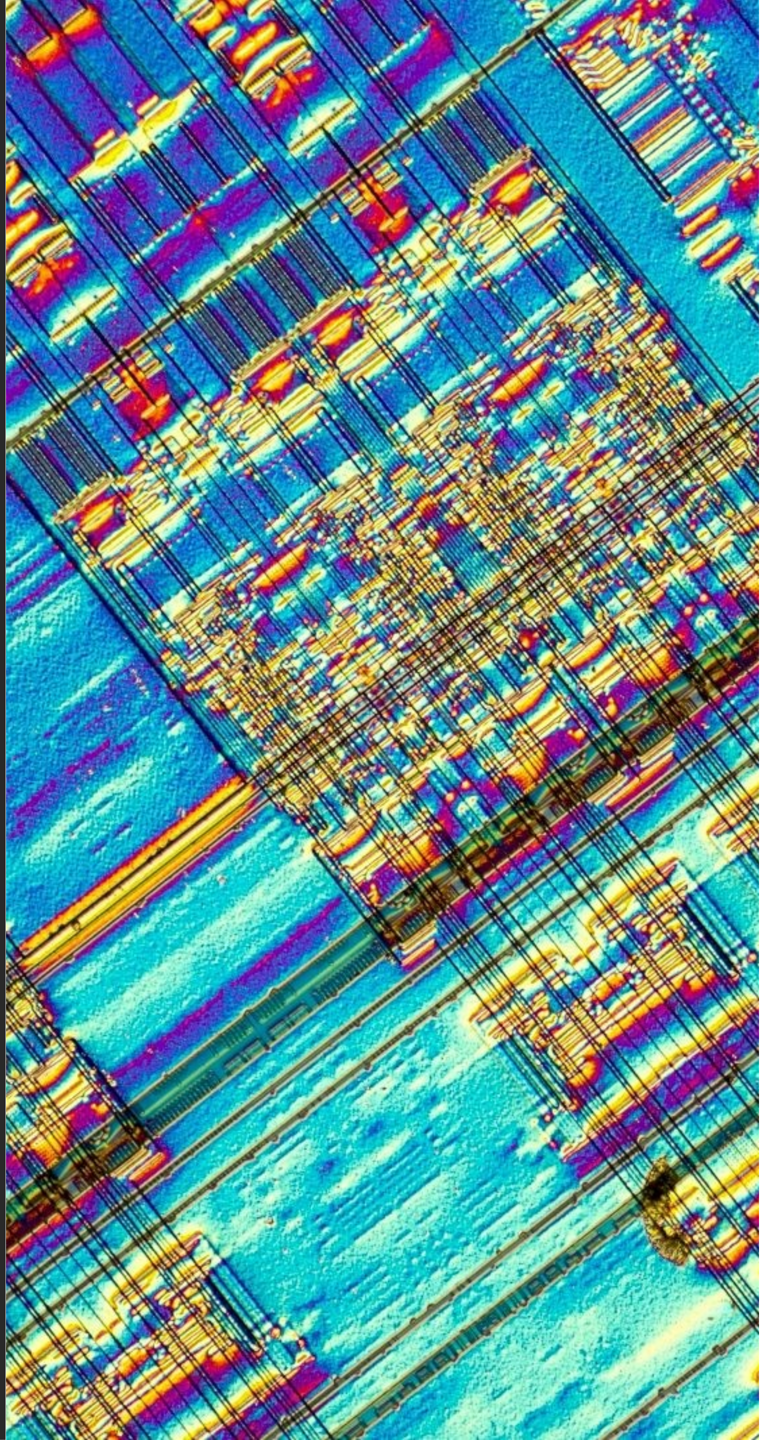
- Hardware limitation may prevent some pages at some addresses to be accessed.
  - Some devices can only perform DMA at certain addresses
  - Some architectures can physically access more memory than they can virtually address (x86_32 for example)
- Zones are a “logical” layout - hardware itself knows nothing about it.
- Memory allocation is not restricted - Linux can fulfill requests from different zones at any time, depending on memory usage.
- Zones are not used for every architecture

# Memory zones

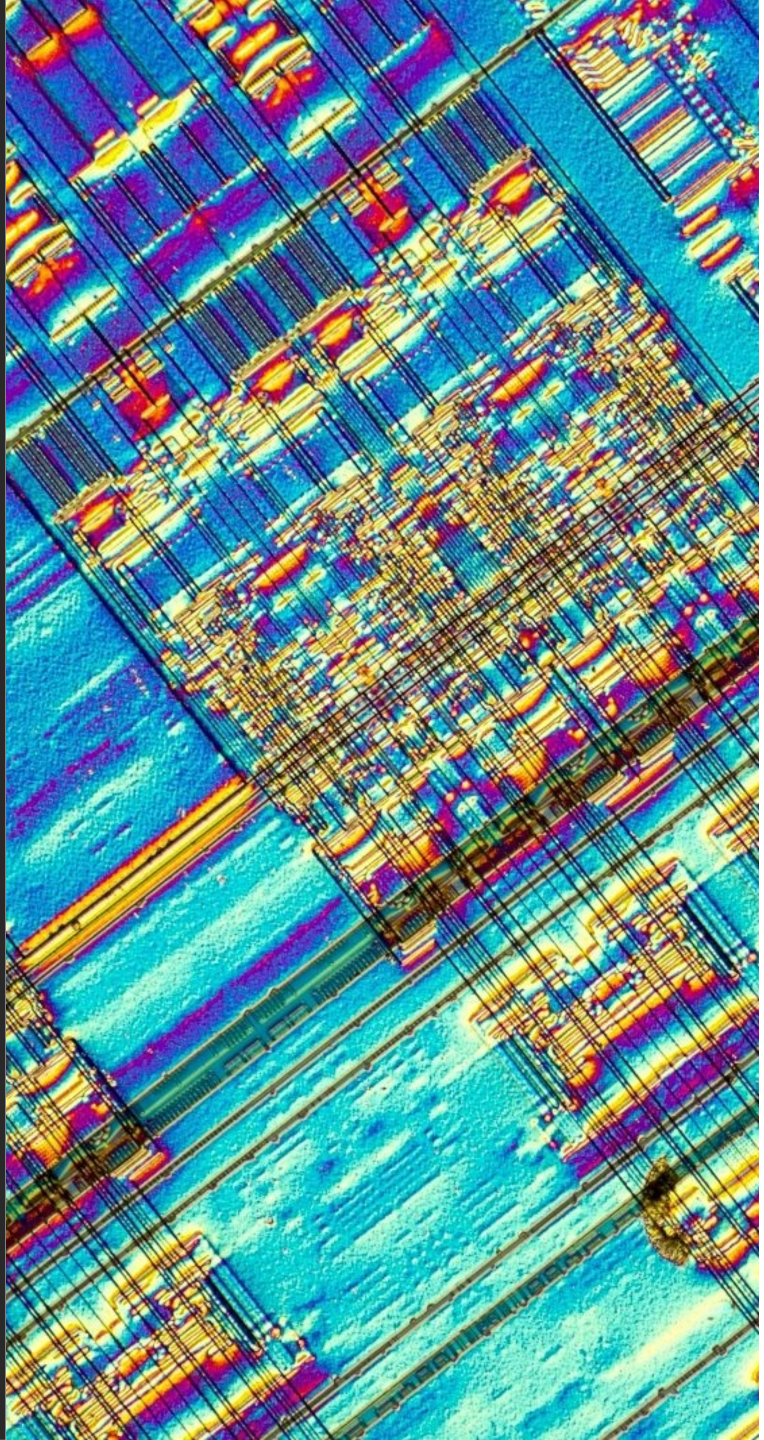
- DMA
- Normal
- High Mem



Referenced in include/linux/mmzone.h



# Linux's memory management APIs



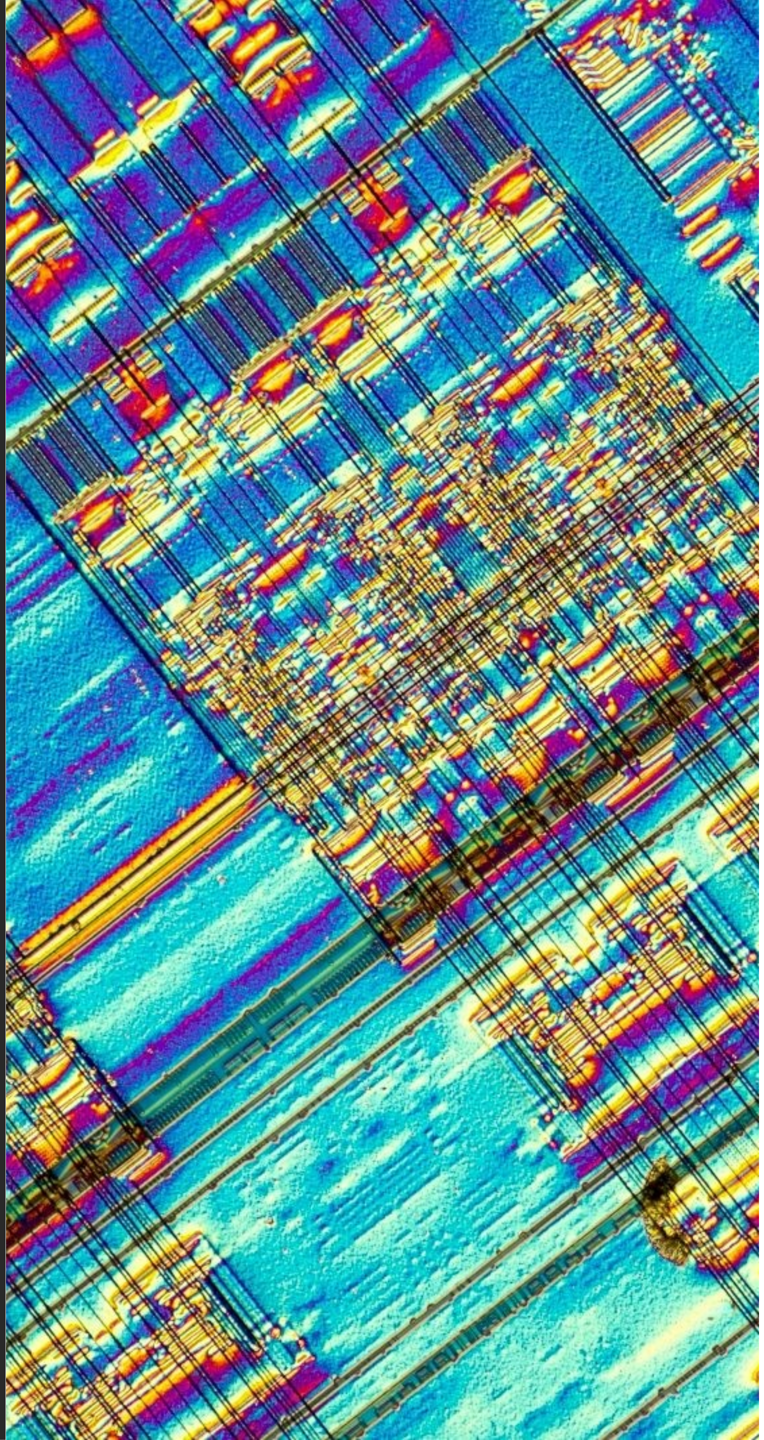
# Page allocation

# Allocating physical pages

- Physical pages within kernel can be directly allocated using the following mechanisms
  - alloc_page(), alloc_pages()
  - page_address()
  - __get_free_page(), __get_free_pages(), __get_zeroed_page()
  - __free_pages(), free_pages(), free_page()

# Allocating physical pages #2

- Pages are always allocated in page-size aligned granularity.
  - E.g - x86 architecture uses multiples of 4096 Bytes
- Allocated pages must be freed once you are done with them.
- Differently from user-space, the Kernel trusts itself, therefore:
  - There are no memory protection mechanisms
  - Kernel will happily let you free pages you didn’t allocate yourself
    - So, make sure you are freeing the right page(s)



# General (byte-sized) memory allocation APIs

# Generic memory allocation

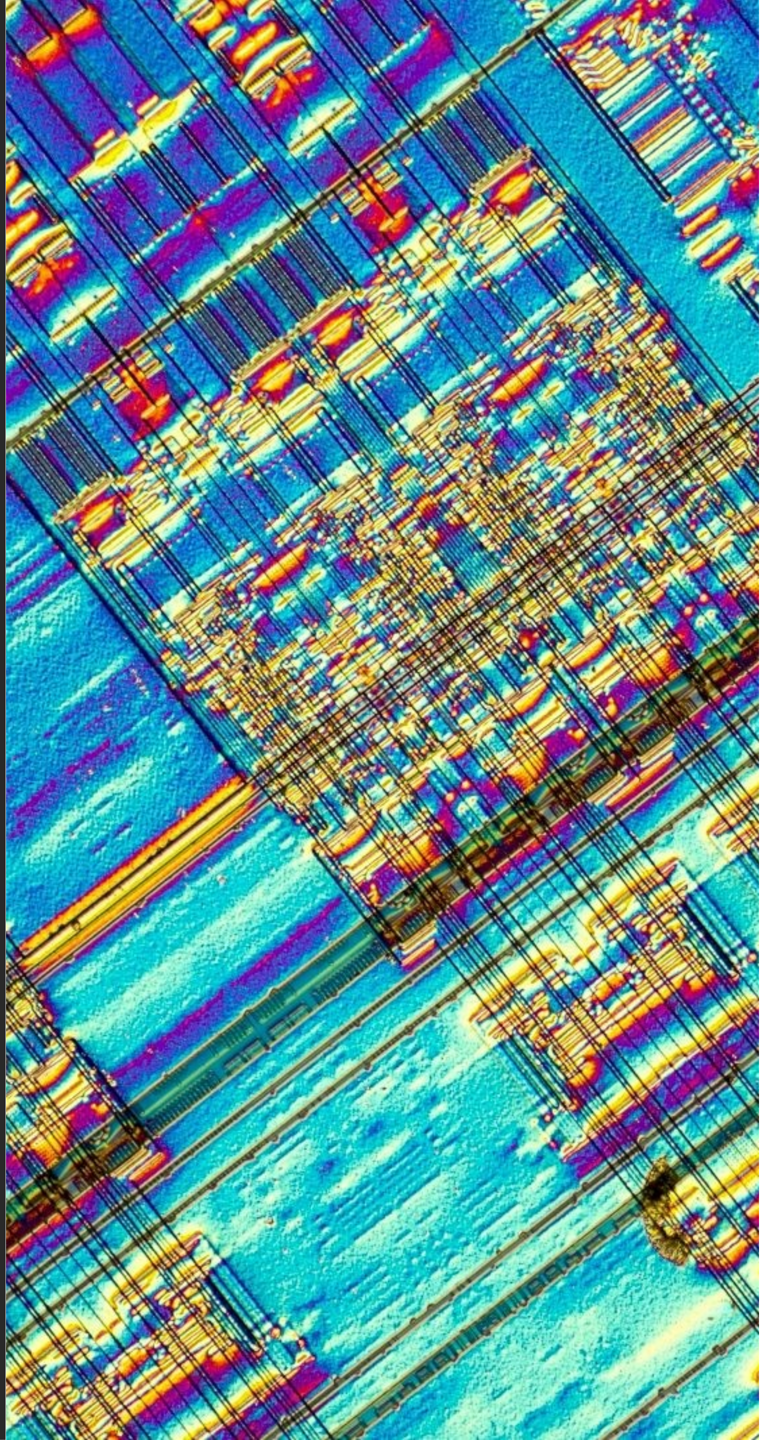
- Most of the time, we don’t need to deal with physical pages directly
- So, the kernel provides a few ways to virtually allocate memory in byte-size chunks
  - Those mechanisms still manipulate physical pages under the hood though.

# vmalloc() - vfree()

- Can be used to allocate a virtual memory region with a byte-size granularity
- Most flexible way to allocate memory within the kernel, because
  - Allocated regions are only virtually contiguous
  - There is no guarantee it will be physically contiguous too.
- Usually, only hardware devices require physically contiguous memory

# vmalloc() - vfree() #2

- Because vmalloc()'ed memory is only virtually contiguous:
  - It  requires the allocator to setup page tables, thich results in TLB thrashing, so
  - vmalloc() is more expensive, might not be a good option when performance is a must.
- On the other hand, with memory fragmentation, large contiguous regions of memory becomes rare, so vmalloc() is a good alternative for large chunks of data
- As any memory, vmalloc()'ed memory should also be freed



# SLAB caches

# SLAB, SLOB, SLUB

- Up until Linux 6.8, we had three different implementations of the SLAB cache.
    - SLAB, SLOB and SLUB
- Everything but SLUB got removed from Linux in 6.8
- Now we have a single implementation of the SLAB cache, using the SLUB implementation.
- DO NOT CONFUSE **SLAB Cache** with its **SLUB** implementation.

# What are SLABs?

- Slabs are “pools” of pre-allocated memory regions of a specific size and/or data type
- Whenever we need to allocate a new object, such object is already allocated
  - We save time with memory allocation
- This is doable for example, by allocating many objects at once, and using a list of free objects to track them down… So, why a generic layer?
  - The kernel memory allocator wouldn’t be aware of this list usage so that it couldn’t fine control it.
  - We don’t need to keep reinventing the wheel

# What are SLABs? #2

- The Linux kernel provide a generic interface for that, known as **SLAB Cache**
- The SLAB cache attempts to leverage a few principles:
  - Frequently used data structures tend to be allocated/freed often
  - Frequent alloc/dealloc results in memory fragmentation over time
  - Memory alloc/dealloc are costly operations

# What are SLABs? #3

- By using a generic layer, and centralizing memory allocation within the slab layer, the kernel is aware of the usage of each slab cache, so it can:
  - Be aware of total cache and objects size
  - Shrink caches by freeing unused objects when needed (like a low-memory scenario)
  - Create per-processor caches, so allocations can be performed without a SMP lock
  - Stored objects can be configured to prevent multiple objects mapping to the same cache lines

# SLAB cache usage examples

- Inode structs
- task_struct structs
- Almost everything inside kernel, that doesn't need to deal with physical memory directly.

# SLAB caches organization

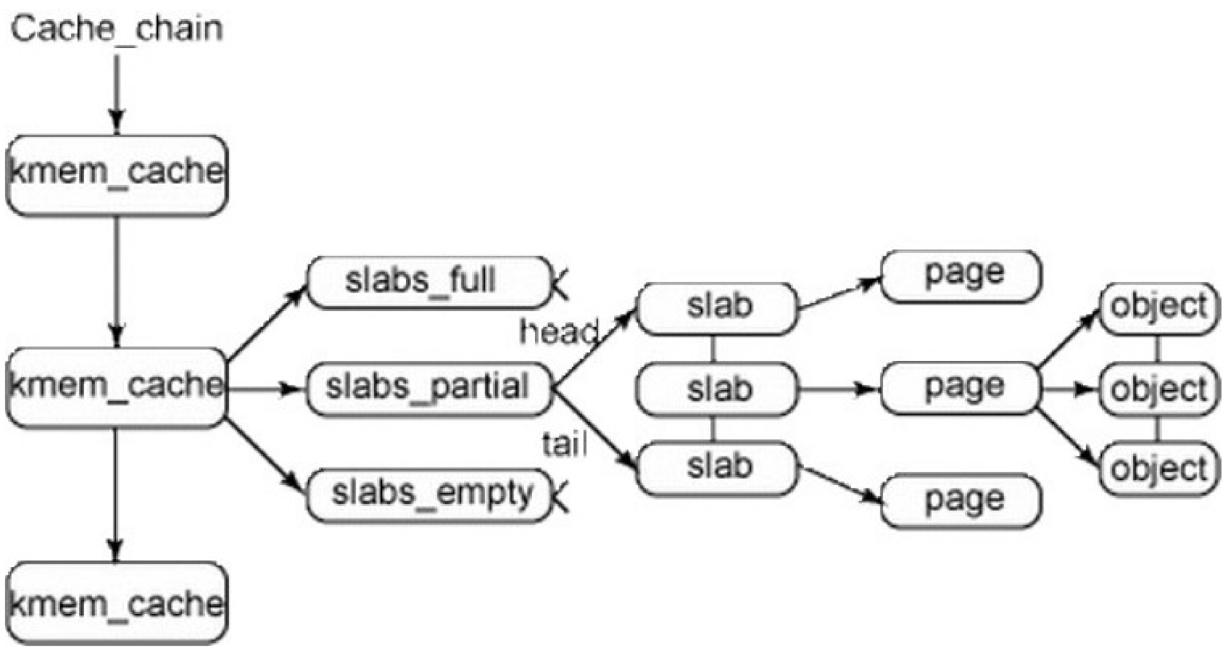
- Each cache is split into different **“slabs”**
- Each slab can be in three states:
  - full - partial - empty
- New allocation requests are attempted to be satisfied from a partially filled slab (if one exists).
  - Fallback to an empty slab
    - Fallback to allocate a new slab and new objects within that slab

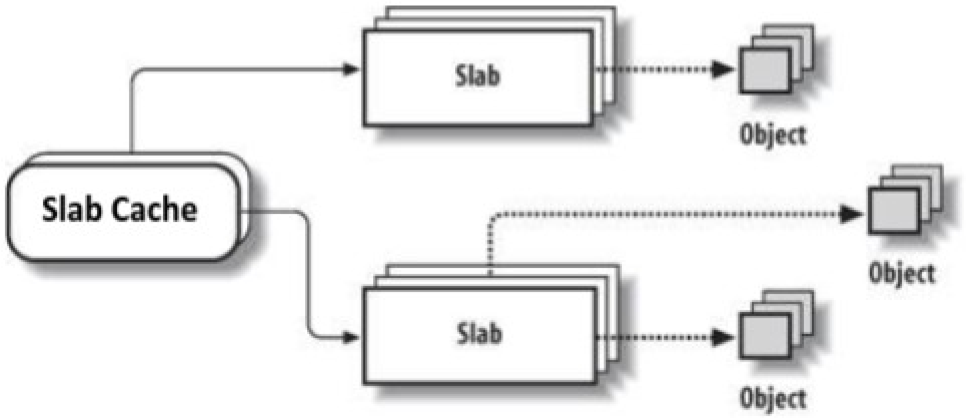


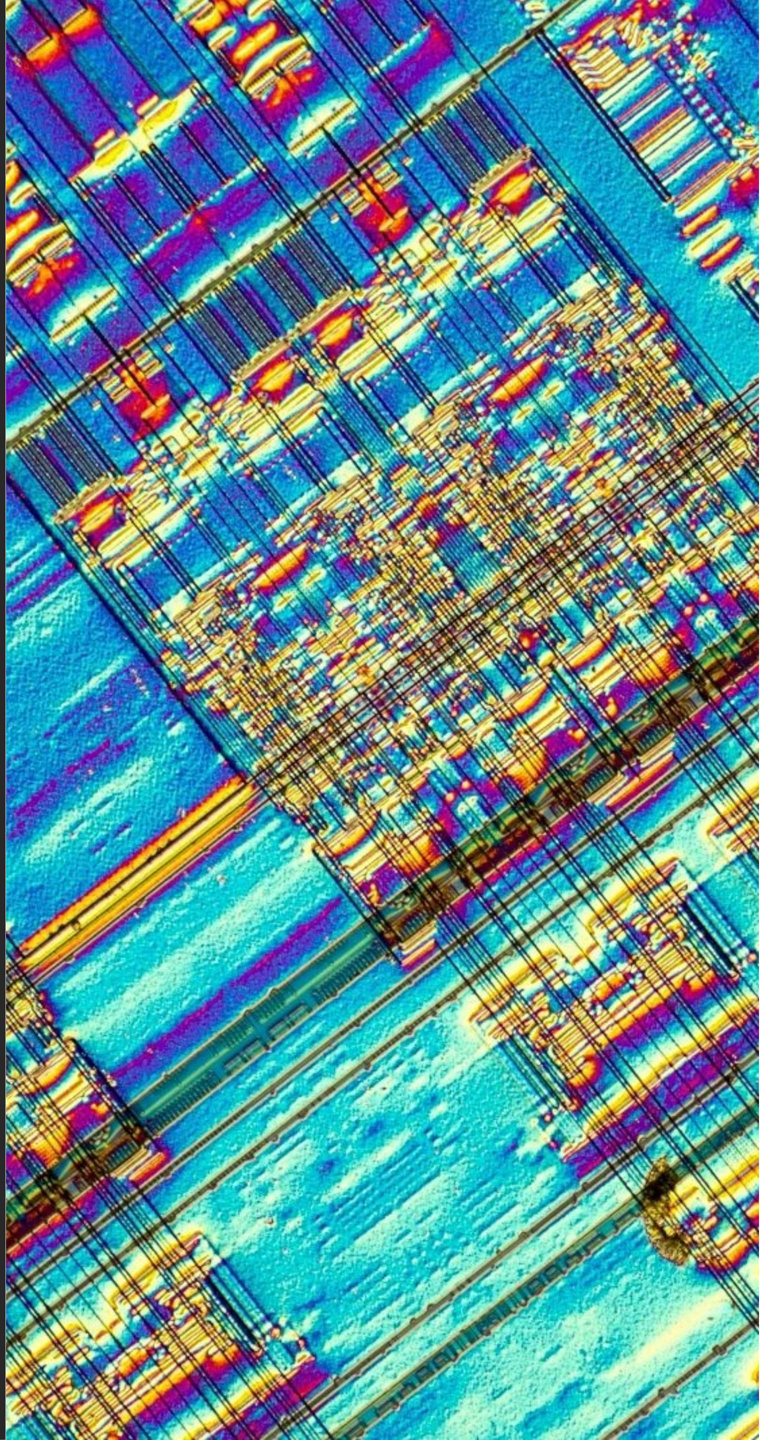
Red Hat

# SLAB caches organization

- slab cache drawing





# SLAB cache APIs

# Dealing with slab cache

- Creating a new slab cache:
  - kmem_cache_create() - kmem_cache_destroy()
    - Behavior can be controlled using some flags
- Allocating objects from a specific cache:
  - kmem_cache_alloc()/kmem_cache_zalloc() - kmem_cache_free()

# kmalloc() - kfree()

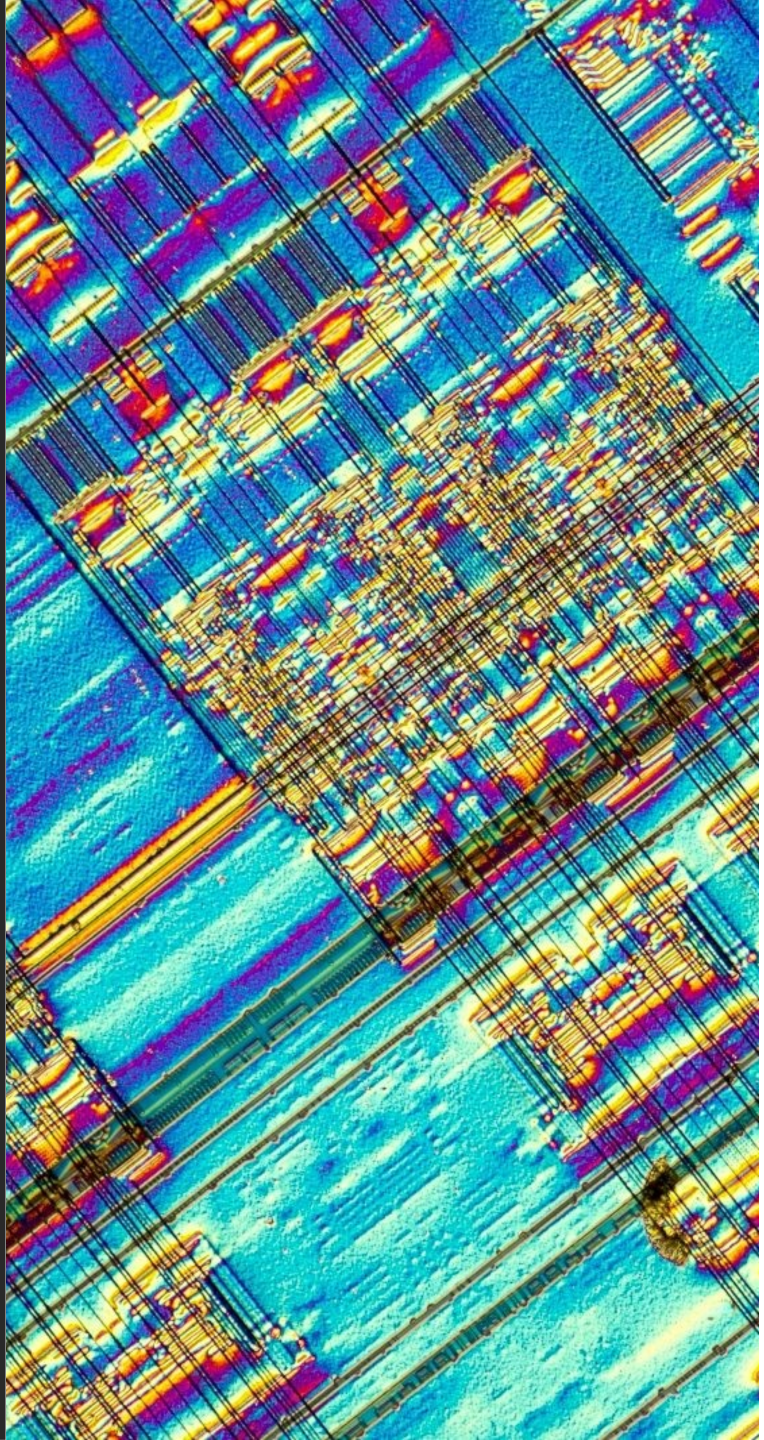
- The ‘default’ memory allocation mechanism for objects smaller than PAGE SIZE
- Similar behavior to userspace malloc()/free() with a few particularities
  - The flags parameter
  - The amount of memory that can be allocated, is limited.
  - Memory allocated is **physically contiguous**

# kmalloc() - kfree() #2

- The amount of memory kmalloc() can allocate is limited, usually 2*PAGE_SIZE
- kfree() - free the regions allocated by kmalloc()
  - Again, kernel will happily let you kfree() random regions of memory.
- kmalloc() is actually a generic abstraction of the slab layer
  - Under the hood, kmalloc() actually works by allocating ‘generic objects’ in a slab cache

# kvmalloc() - kvfree()

- kmalloc() with a vmalloc() fallback
- It tries to allocate physically contiguous memory with kmalloc()
  - If it fails, it fallback to vmalloc() allocation
- Good alternative if you need memory at all costs and can for trade performance.
  - And yet, it still can fail
- kvfree() - Free the memory region by type checking the kind of allocation that has been done



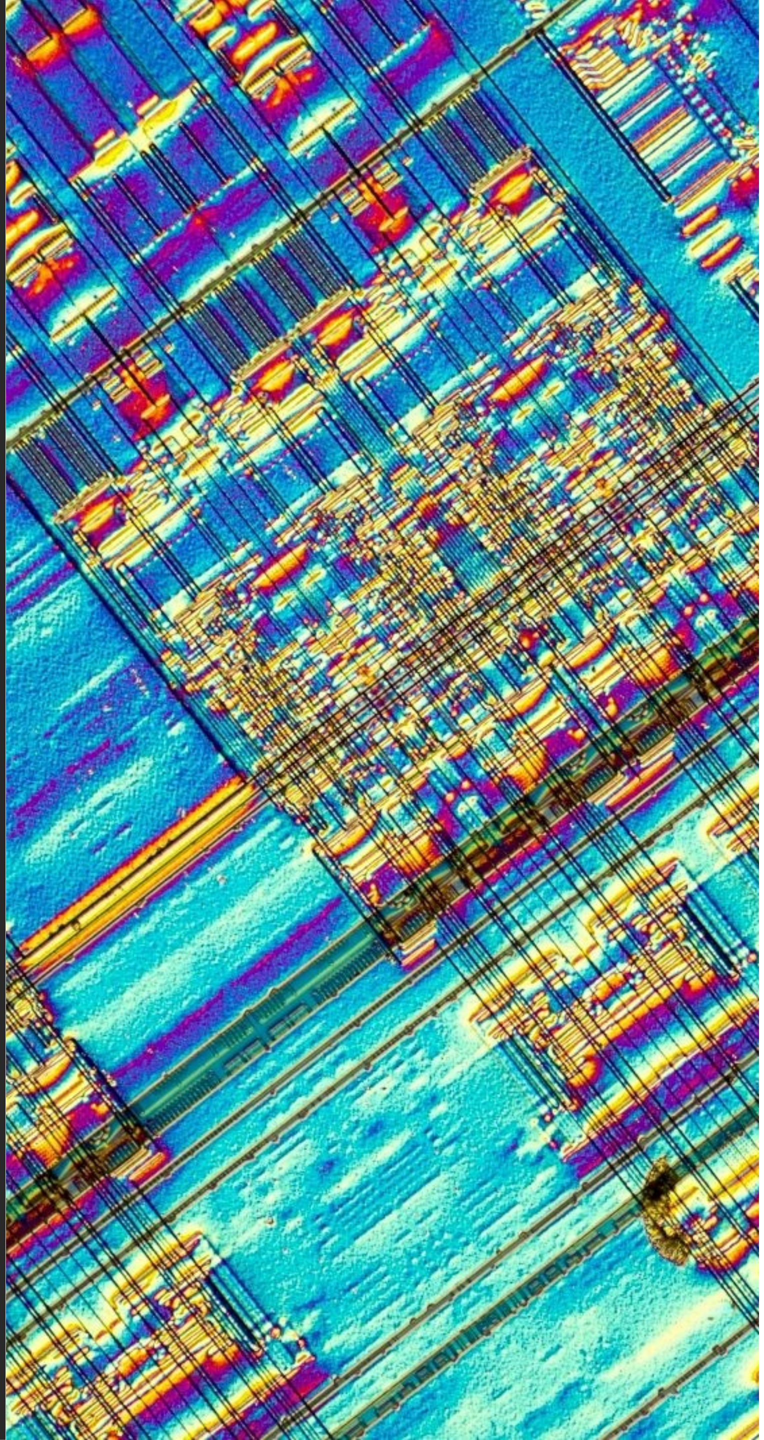
# GFP Flags

# Controlling the memory allocator

- Allocating memory within the kernel is a bit more complicated
- Memory allocation might trigger unwanted or unexpected side-effects, like
  - Generate disk I/O to reclaim memory
  - Generate filesystem operations
  - Allocated memory is in a different region and a device can’t access it for DMA
- The memory allocator in Linux, can be controlled using the **Get Free Pages** (GPF) flags

# GFP flags

- GFP flags high-level categories
  - Zone modifiers - Zone selection
  - Mobility and placement flags - Reclaimable? Can it be migrated?
  - Watermark modifiers - Emergency memory reserves
  - Reclaim modifiers - How kernel can reclaim memory if needed
  - Action modifiers - Use different behaviors
- There are dozen of GFP flags, but most of the time, we will be using the same ones over and over



# Linux Kernel’s stack

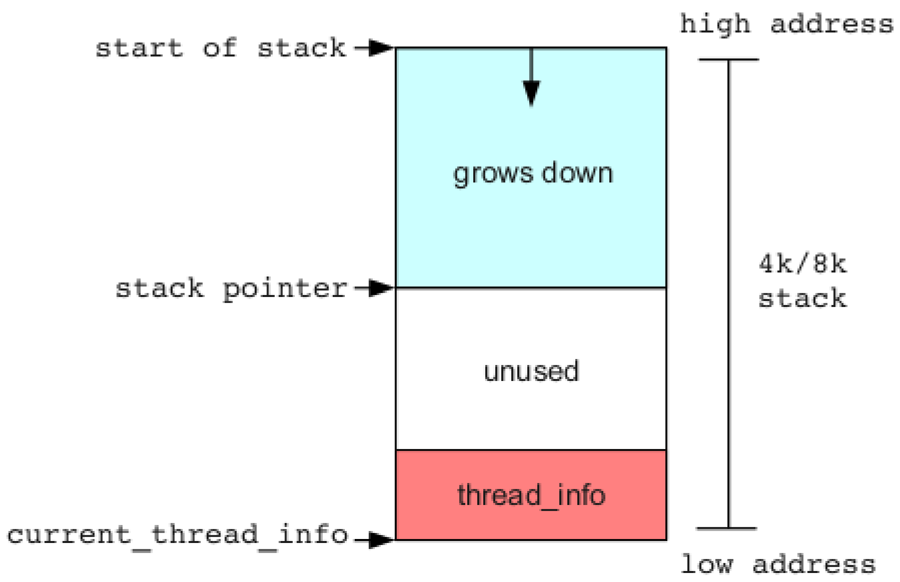
# Stack allocation within kernel

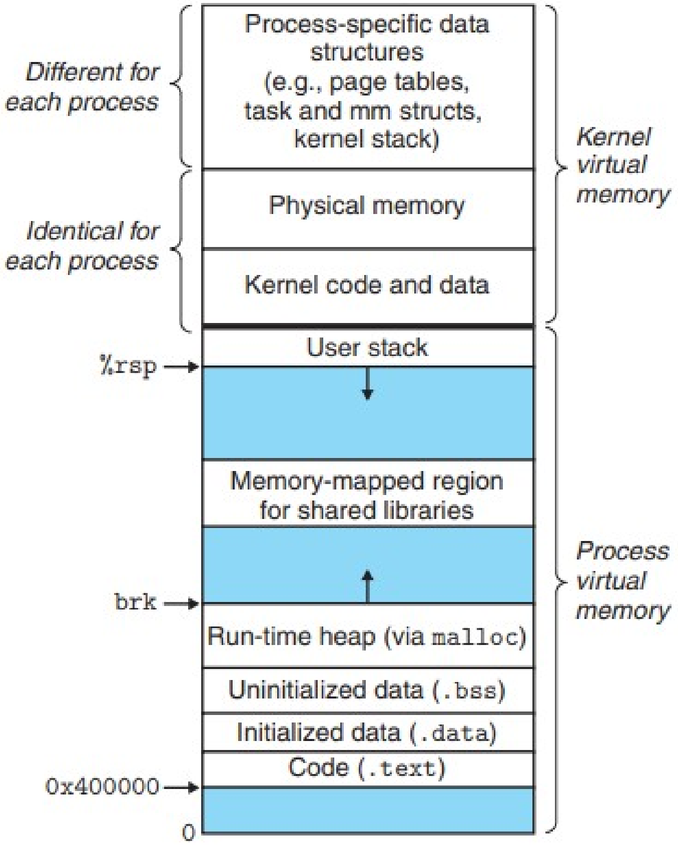
- Different from user-space, the kernel doesn't have the luxury of a dynamically allocated stack.
- The Kernel stack is small and of a fixed size
  - Size is architecture dependent - Usually 2 * PAGE_SIZE
- Linux kernel make very little effort to manage kernel-space processes stacks
  - Overflowing the stack will corrupt whatever data is beyond it (starting with struct thread_info)
- KASAN has interesting options to debug stack overflows



https://www.kernel.org/doc/html/next/x86/kernel-stacks.html

# Linux kernel stack



# Thank you

Red Hat is the world’s leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make
Red Hat a trusted adviser to the Fortune 500.

linkedin.com/company/red-hat

youtube.com/user/RedHatVideos

facebook.com/redhatinc

twitter.com/RedHat



Red Hat