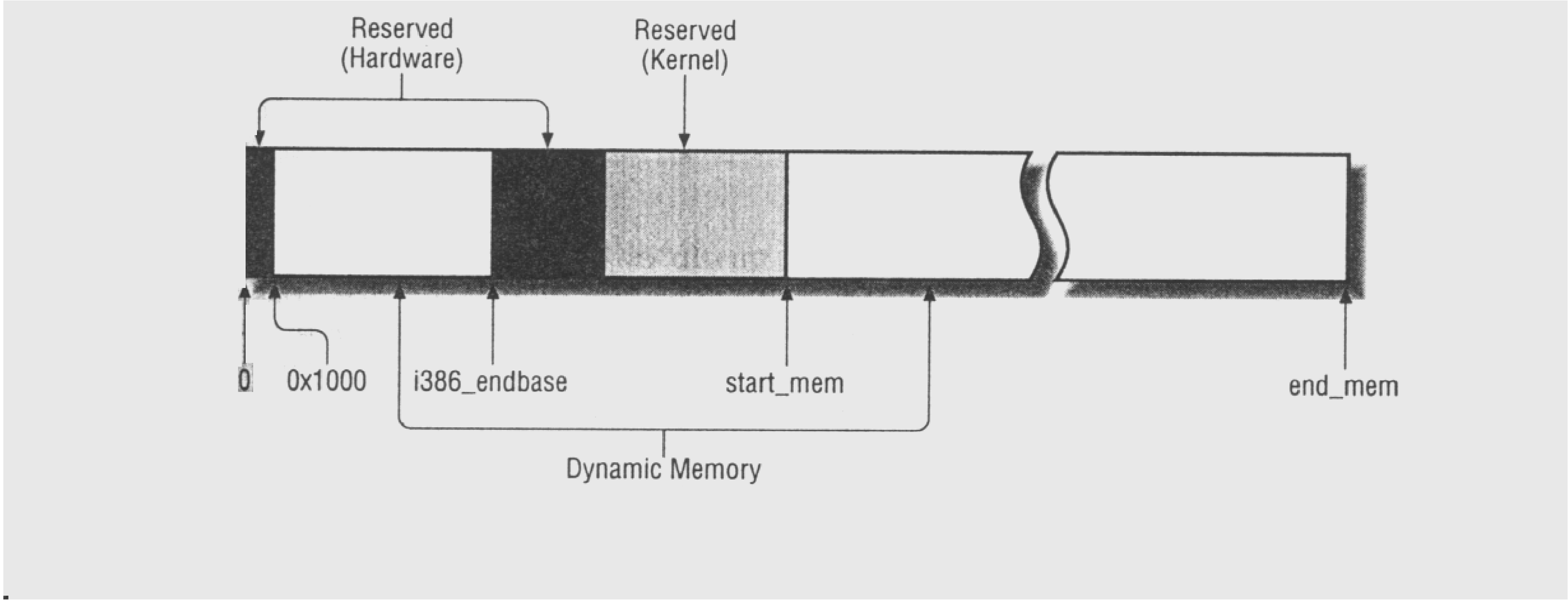
Ausgewählte Betriebssysteme

Memory

Memory Management

- Kernel
  - Page Frames
  - Buddy Allocator
  - Slab Allocators
  - Buffer Cache
  - Page Cache
- Process
  - Memory Regions

Memory Map


Reserved
(Hardware)
Reserved
(Kernel)
0
0x1000
i386_endbase
start_mem
end_mem
Dynamic Memory

Page Frame

- kernel must keep track of state
  - kernel code, page cache, kernel data etc.
- which pages are available
- page descriptor for each frame
  - mem_map_t *mem_map
- linked into appropriate list if needed

Buddy

- robust, efficient kernel allocator
- contiguous page frames
- external fragmentation
  - paging
  - managing pages in a suitable way

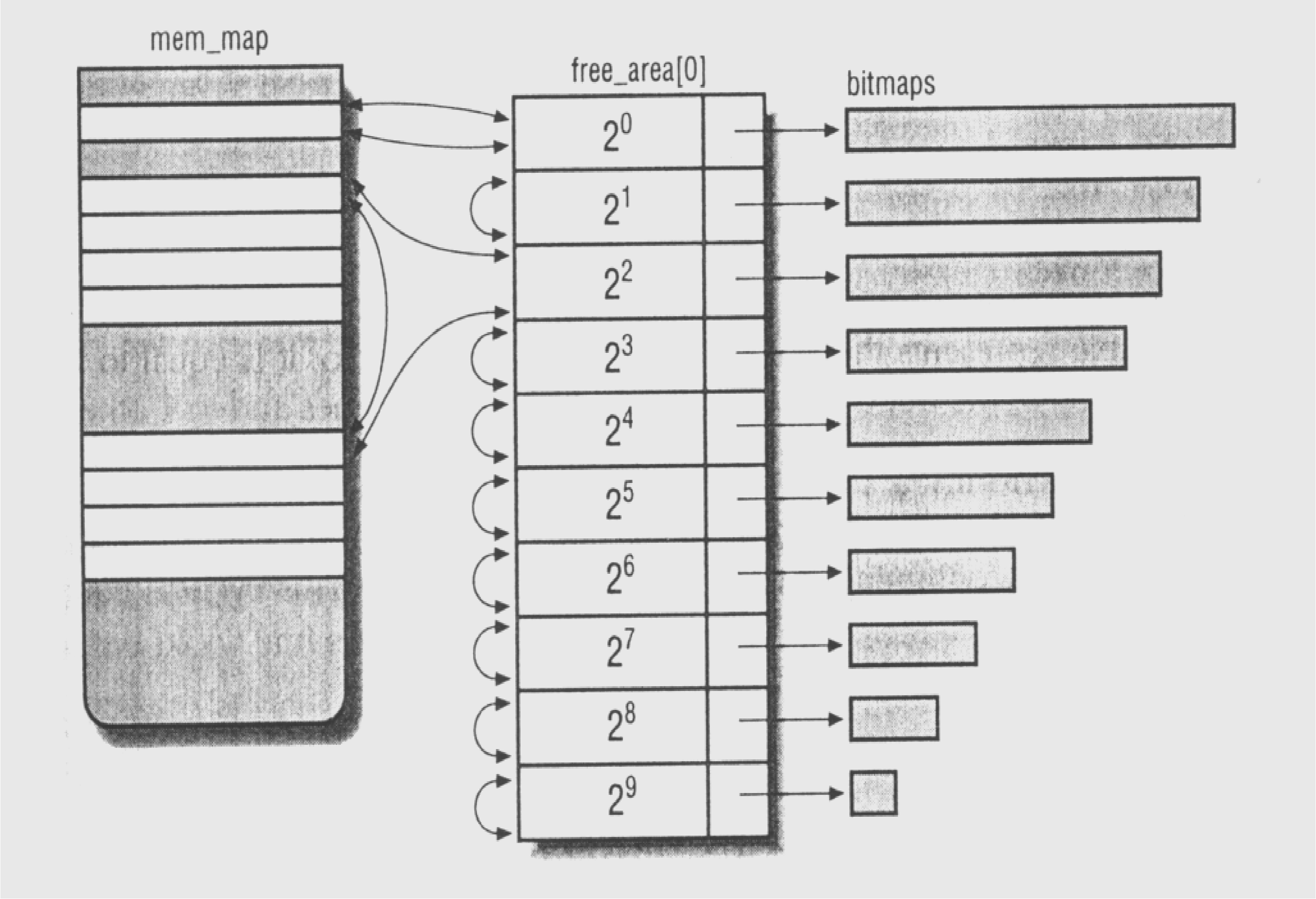
Why not paging

- sometimes physical contiguous pages required
  - DMA bypasses CPU paging circuitry
- paging modification deteriorate TLB efficiency
  - TLB flushing required for consistency

Buddy Allocator

- well-known buddy system algorithm
- free pages are grouped into 10 lists
  - 1 .. 512 contiguous pages
- apropriately aligned

Buddy in action



Buddy API

- get_free_page(pfp_mask);
- __get_free_pages(gfp_mask,order);
- free_page(addr);
- free_pages(addr,order);

Memory Area Management

- contiguous physical addresses
- arbitrary length (not necessarily multiple of page size)
  - feq tens or hundreds of bytes
- internal fragmentation
- 2.0 buddies for small requests
  - geometrically distributed size
  - not more than 50 % loss
- 2.2 Slab Allocator
  - first 1994 Solaris 2.4

Slab

- memory areas as objects
  - set of data structures
  - constructor and destructor
  - not used in Linux
- tendency of requesting and releasing same memory type repeatedly
  - e.g. process creation
  - keep memory in cache as long as possible

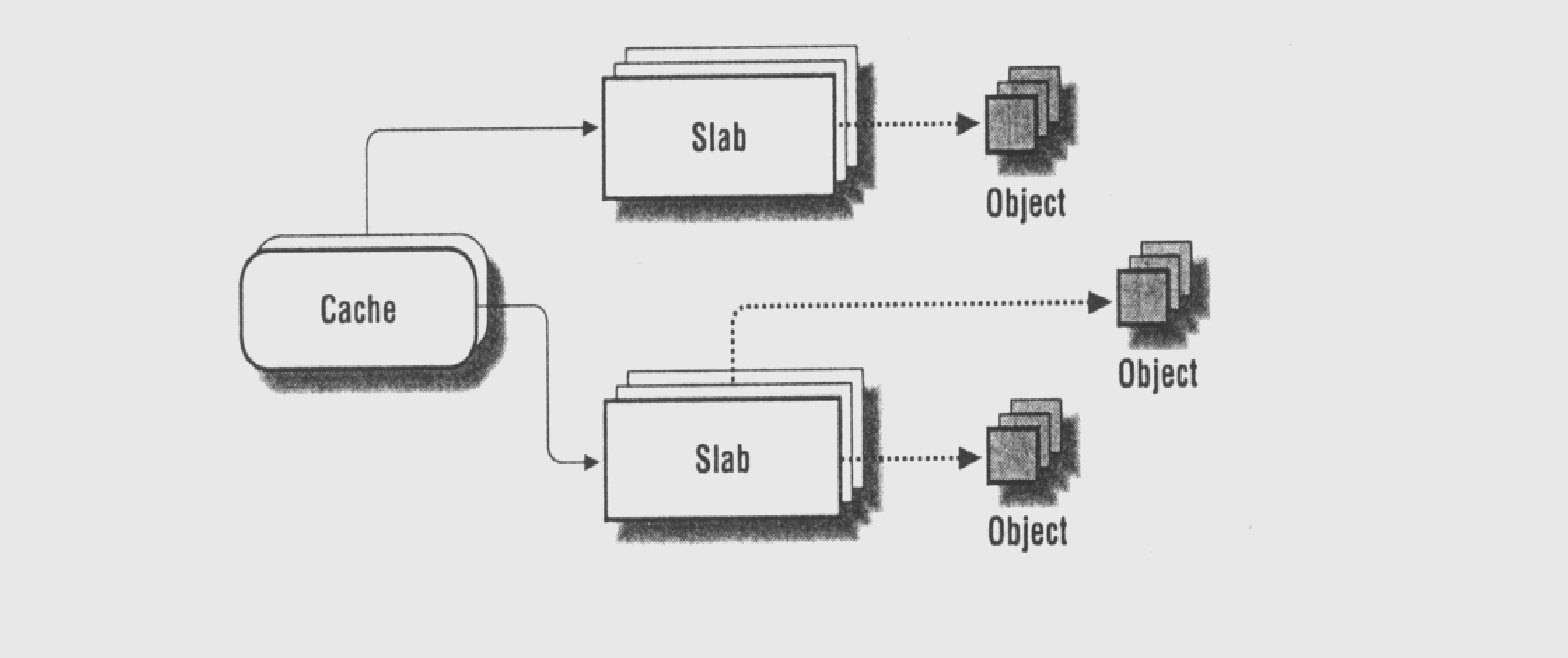
Slab

- if size not geometrically distributed, addresses are less prone to concentrate on physical addresses whose values are power of 2
  - better hardware cache usage
- frequent calls to the buddy allocator pollutes the cache

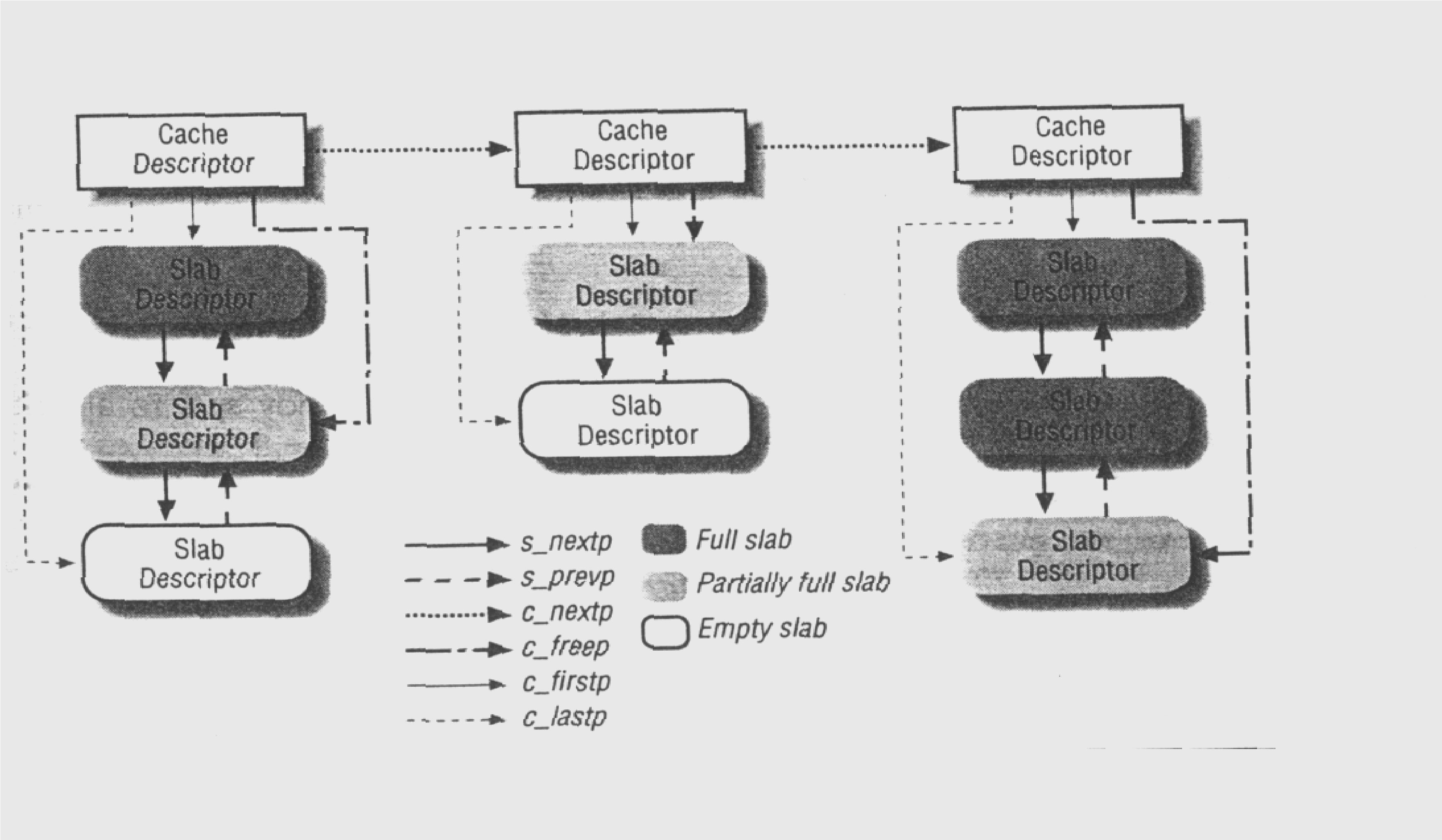
Caches

- object of same kind are stored in caches
  - e.g. file object upon open system call is stored in cache *filp* (file pointer)
  - /proc/slabinfo
- consist of several *slabs*
  - each slab consist of one or more contiguous page frames

Cache



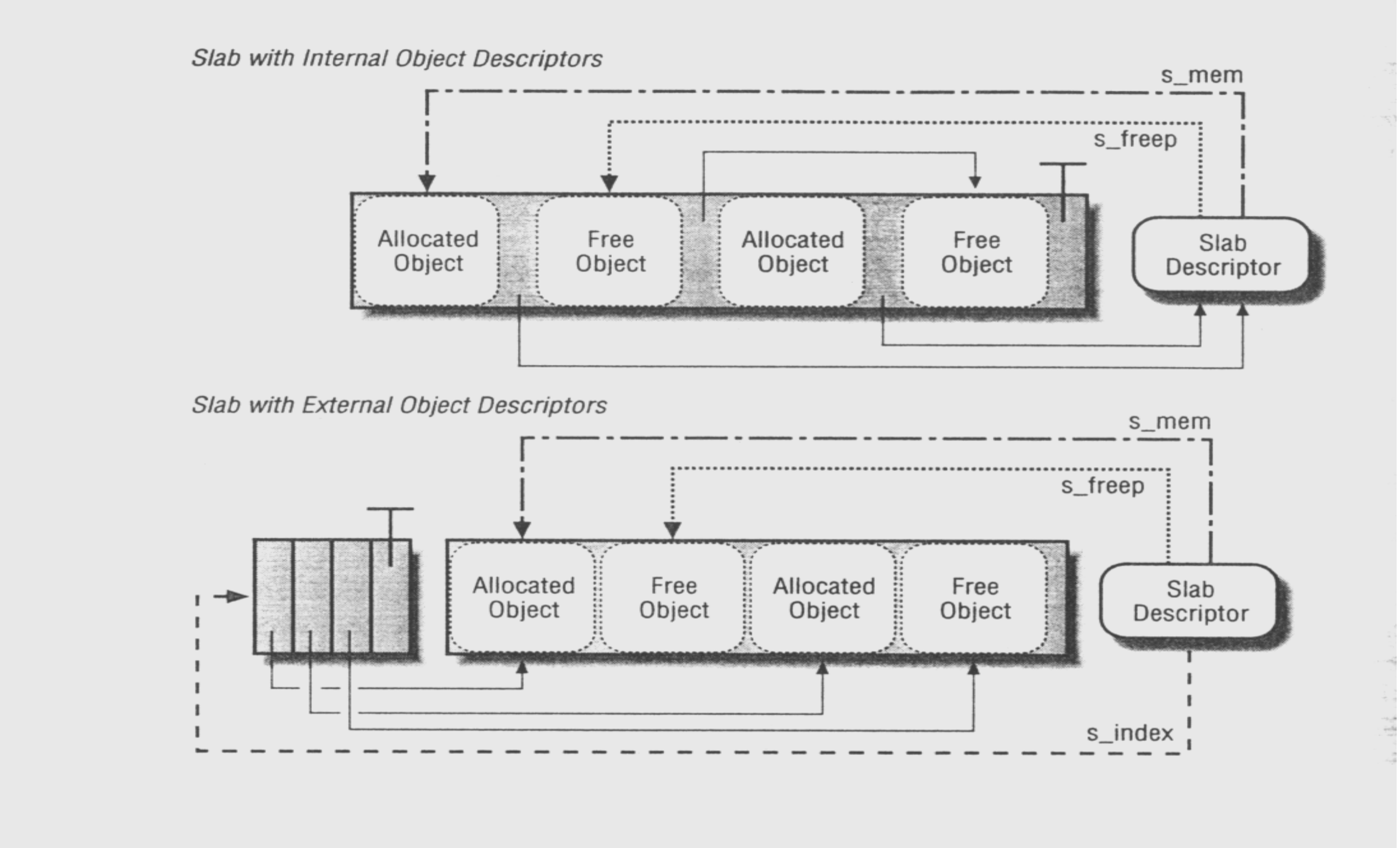
Caches and Slabs



General and Specific Caches

- general
- used only be the slab allocator for own purposses
    - cache descriptors (`cache_cache`)
    - slab descriptors (`cache_slabp`)
    - 13 caches for geometrically distributed memory areas
  - `kmem_cache_init() ,kmem_cache_sizes_init()`
- specific
  - `kmem_cache_create()`

Slabs and Objects

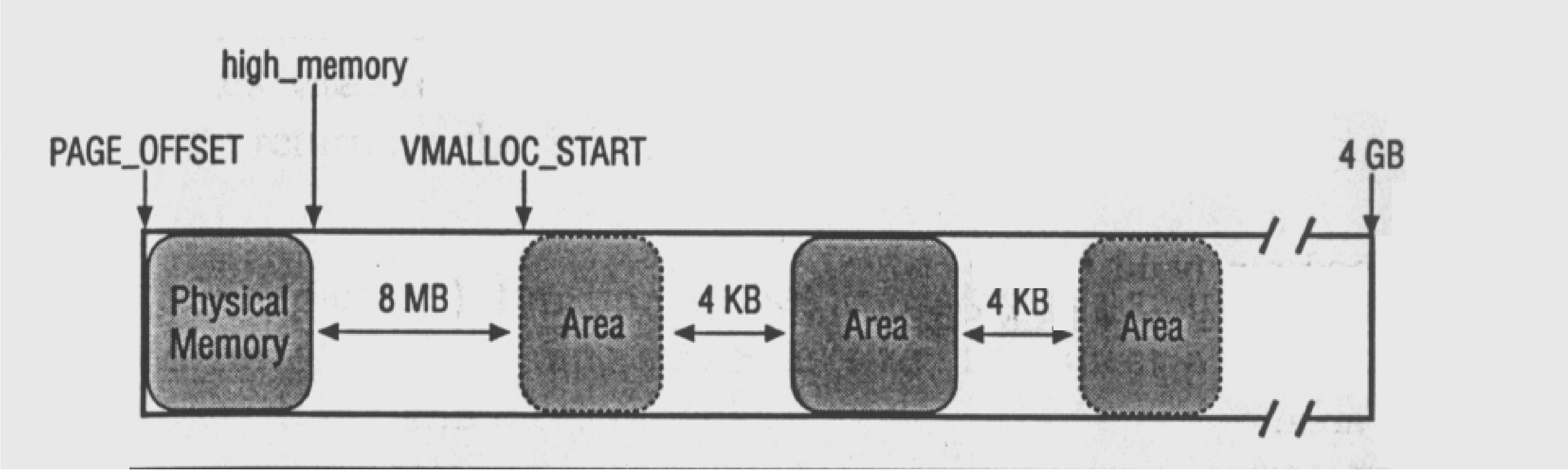


Noncontiguous Memory

- rarely used
- only for (hopefully) infrequent changed objects
  - data structures for active swap areas
  - space for modules
  - buffers for some I/O drivers

Noncontiguous Memory

- vmalloc
- vfree



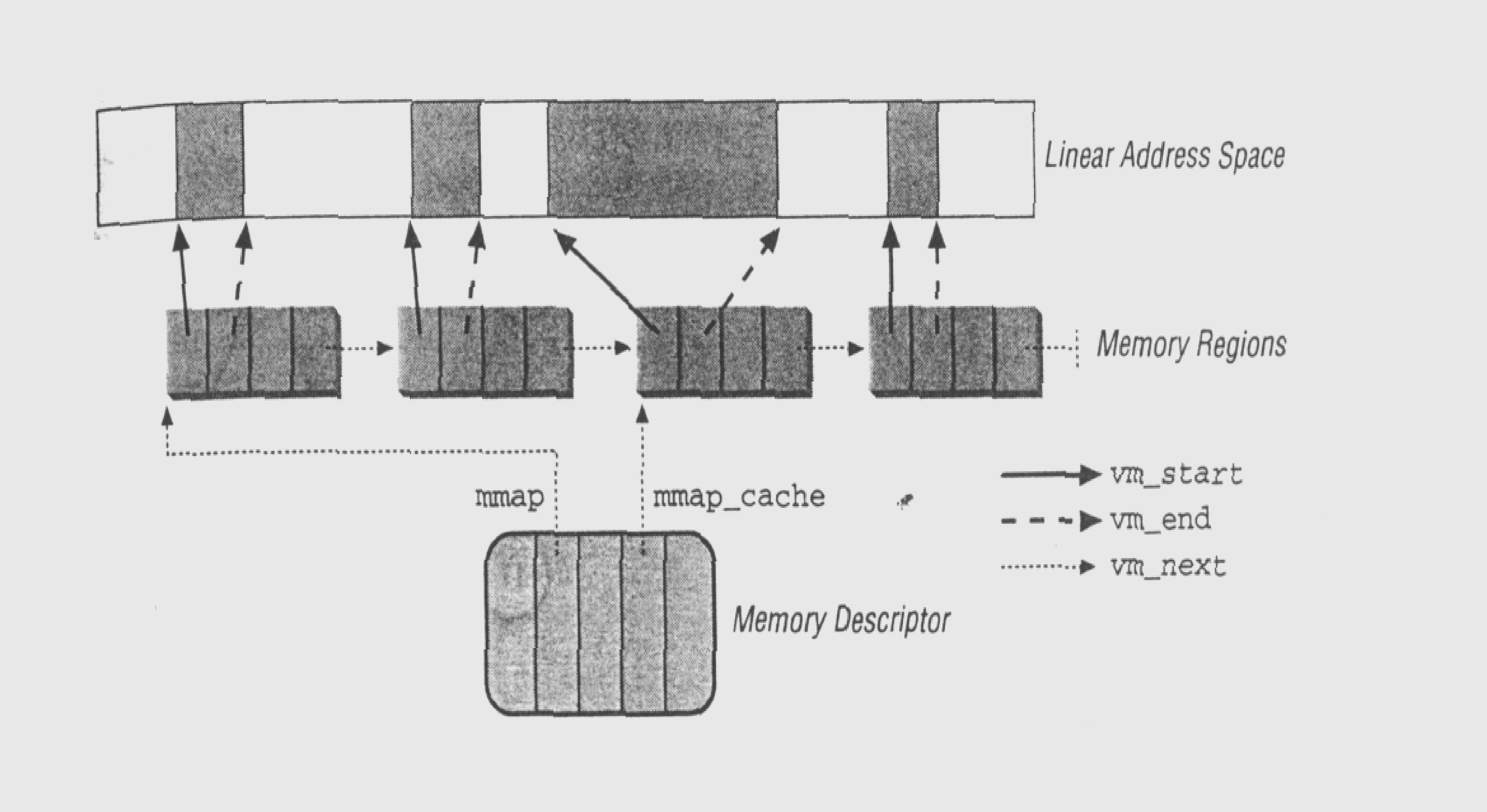
Linux 2.4 and Memory

- Buddies
  - 2.2 has two buddy systems (DMA and Non-DMA)
  - 2.4 adds a third for high physical memory
- Slabs
  - mostly unchanged
  - slab caches can be destroyed
    - modules are expected to do so

Process Address Space

- non-urgent
  - allocation does not mean access
- addressing errors must be caught
- set of linear address
  - memory region
- different access rights
- different for each process
- no relation among processes

Memory Regions



Memory Regions

- Situations for new regions
  - process creation
  - exec
  - memory map
  - stack growth
  - IPC shared memory
  - expand dynamic area (heap)

MM related system calls

- brk
- execve
- exit
- fork
- mmap
- munmap
- shmat
- shmdt

Memory Descriptor

- pointer to regions list
- pointer to Global Directory
- number of allocated pages
- address space size
- reference count
- possibly shared among lightweight processes

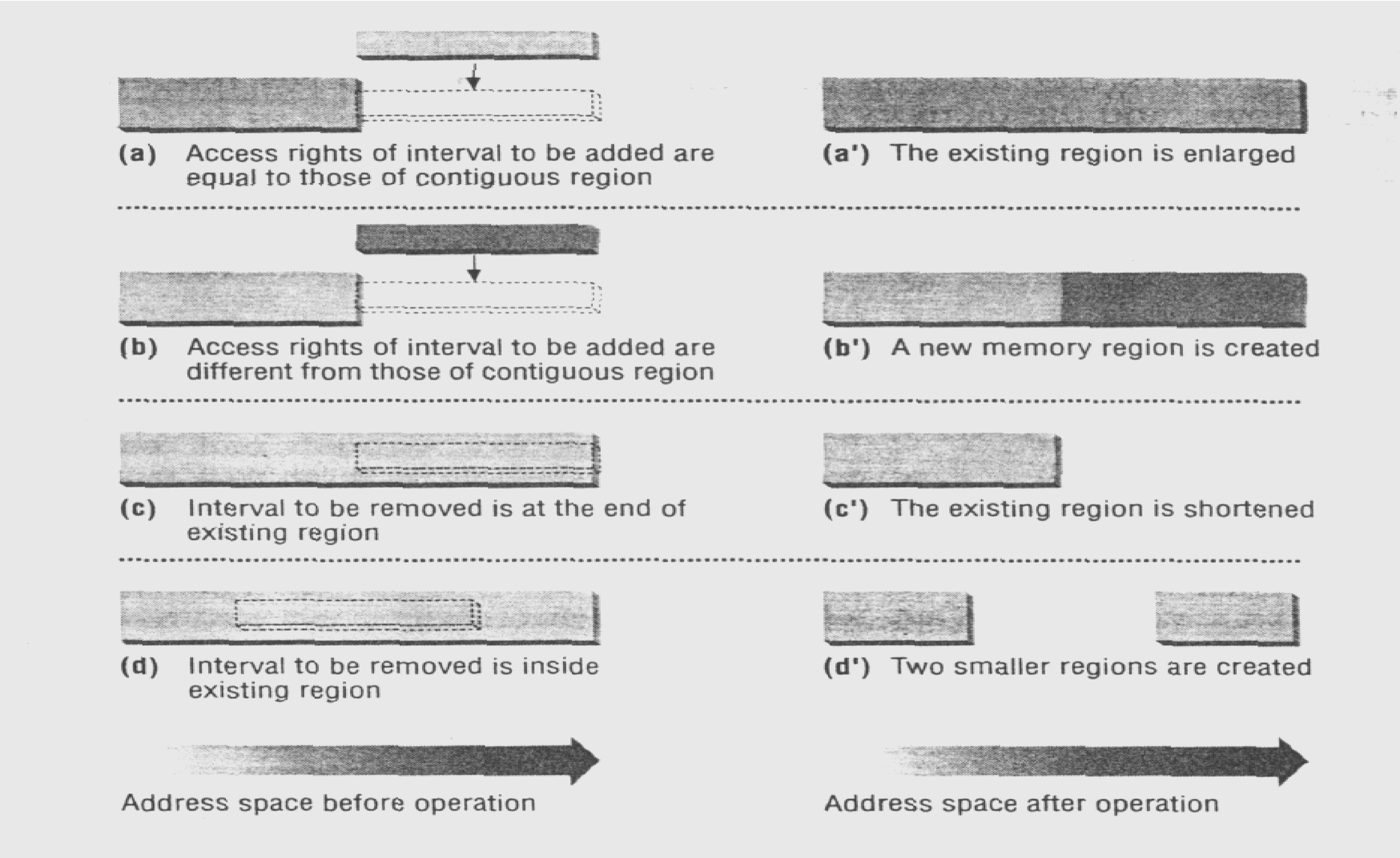
Memory Region

- vm_area_struct
- start of region
- end of region
- access rights
- all regions of a process are linked

Memory Region (2)

- find_vma()
- find_vma_intersection()
- get_unmapped_area()
- insert_vm_struct()
- do_map()
- do_unmap()

Changing Memory Regions



(a) Access rights of interval to be added are equal to those of contiguous region

(a') The existing region is enlarged

(b) Access rights of interval to be added are different from those of contiguous region

(b') A new memory region is created

(c) Interval to be removed is at the end of existing region

(c') The existing region is shortened

(d) Interval to be removed is inside existing region

(d') Two smaller regions are created

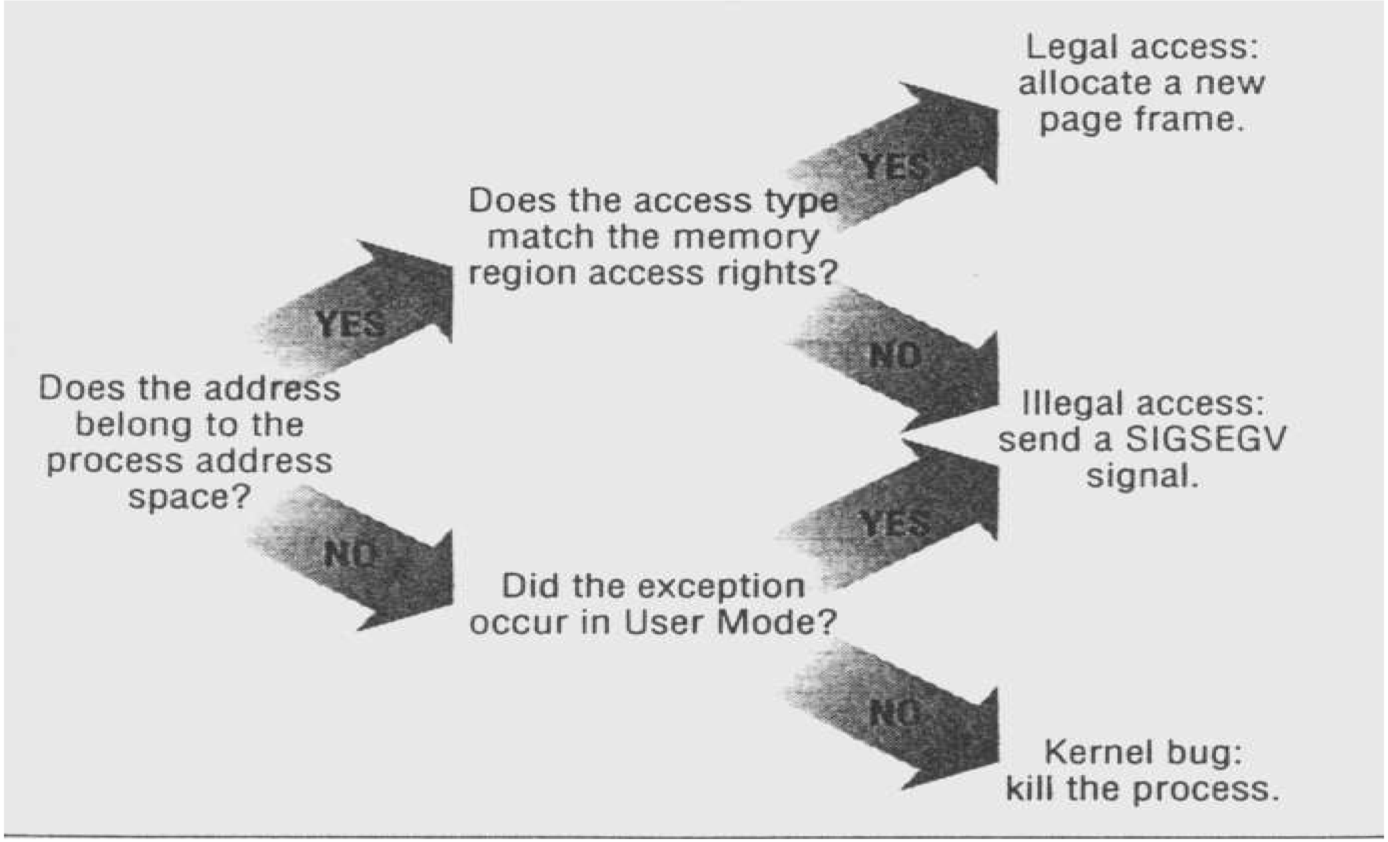
Page Faults

- programming errors
- missing page, though linear address belongs to the process address space
  - contained in some memory region
  - not invalid from process point of view
  - allocate page frame and have process continue

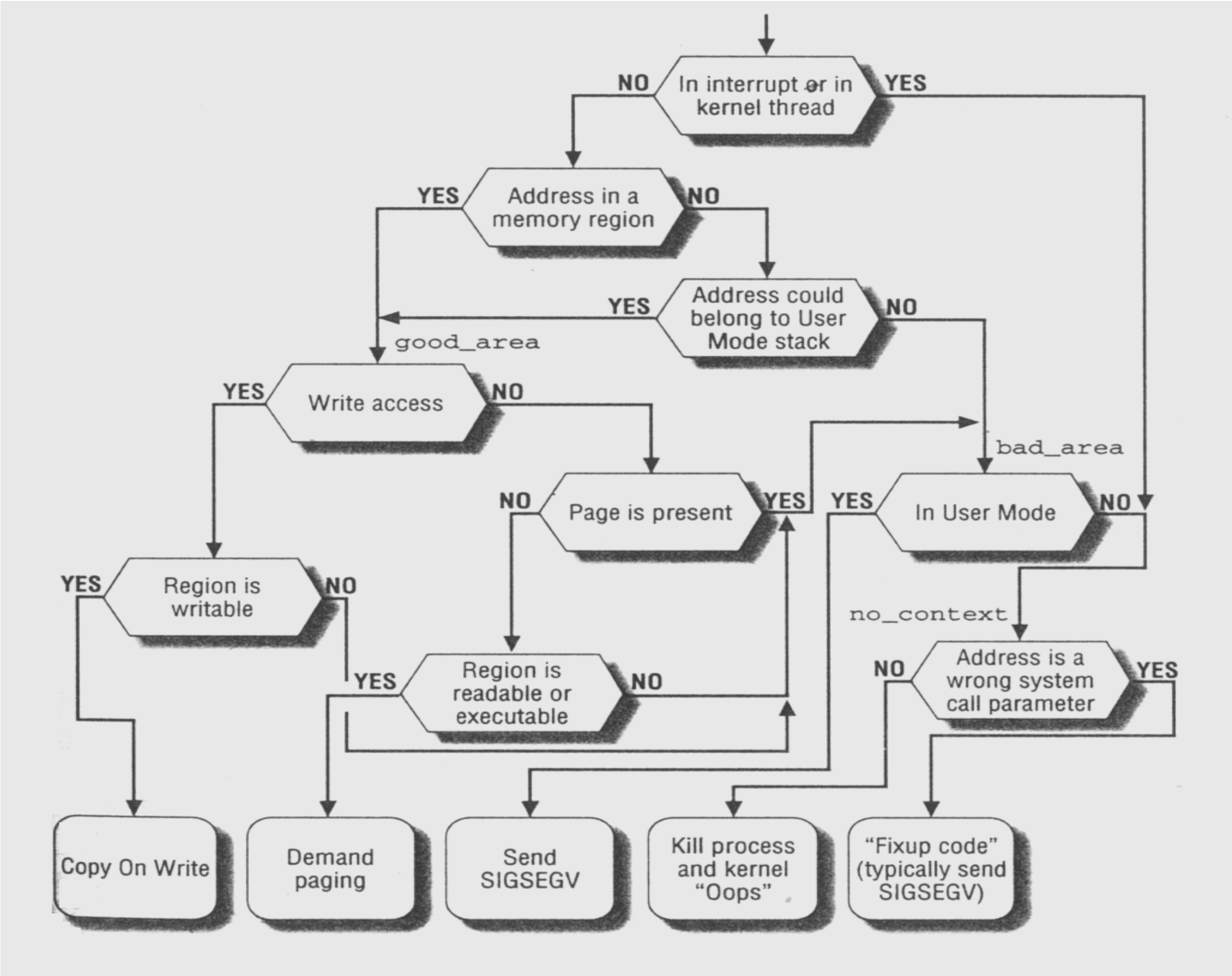
Page Fault

- handle_mm_fault()
  - allocates new pages
  - demand paging
    - do_no_page
      - vma->vm_ops->nopage handler loads page from disk
      - do_anonymous_page()
    - do_swap_page

Page Faults


Does the address belong to the process address space?
YES
NO
Does the access type match the memory region access rights?
Did the exception occur in User Mode?
YES
NO
YES
NO
Legal access: allocate a new page frame.
Illegal access: send a SIGSEGV signal.
Kernel bug: kill the process.

Page Fault (2)


In interrupt or in kernel thread
NO
YES
Address in a memory region
YES
NO
Address could belong to User Mode stack
YES
NO
good_area
Write access
YES
NO
bad_area
Page is present
NO
YES
In User Mode
YES
NO
Region is writable
YES
NO
no_context
Region is readable or executable
YES
NO
Address is a wrong system call parameter
NO
YES
Copy On Write
Demand paging
Send SIGSEGV
Kill process and kernel "Oops"
"Fixup code" (typically send SIGSEGV)

Copy On Write

- share pages
- duplicate on modification attempts
- handle_pte_fault()
  - allocate new page frame
  - adjust counter in frame descriptor
  - Copy content

Creating

- clone(), fork(), vfork()
- copy_mm()
  - copy_segments()
  - new_page_tables()
    - 0-3 GB clear
    - 3-4 GB initialized from swapper process
  - dup_mmap()
    - Duplicate memory regions
    - set up the copy-on-write mechanism

Heap

- C-library for user land
  - malloc, calloc, free, brk
  - only brk as system call
- brk syscall
  - check if request overlaps with current regions
  - maps/unmaps page

Disk Caches

- try to keep as much as possible in memory
- Buffer Cache
  - cache for buffer I/O operations
  - blocks of block devices
- Page Cache
  - content of files
  - not necessarily adjacent on disk

Operations related to disk caches

I/O Operation	Cache	System Call	Kernel Function
Read a block device file	Buffer	read()	block_read()
Write a block device file	Buffer	write()	block_write()
Read an Ext2 directory	Buffer	getdents()	ext2_bread()
Read an Ext2 regular file	Page	read()	generic_file_read()
Write an Ext2 regular file	Page, Buffer	write()	ext2_file_write()
Access to memory-mapped file	Page	None	file_map_nopage()
Access to swapped-out page	Page, Buffer	None	do_swap_page()

Finding Buffers

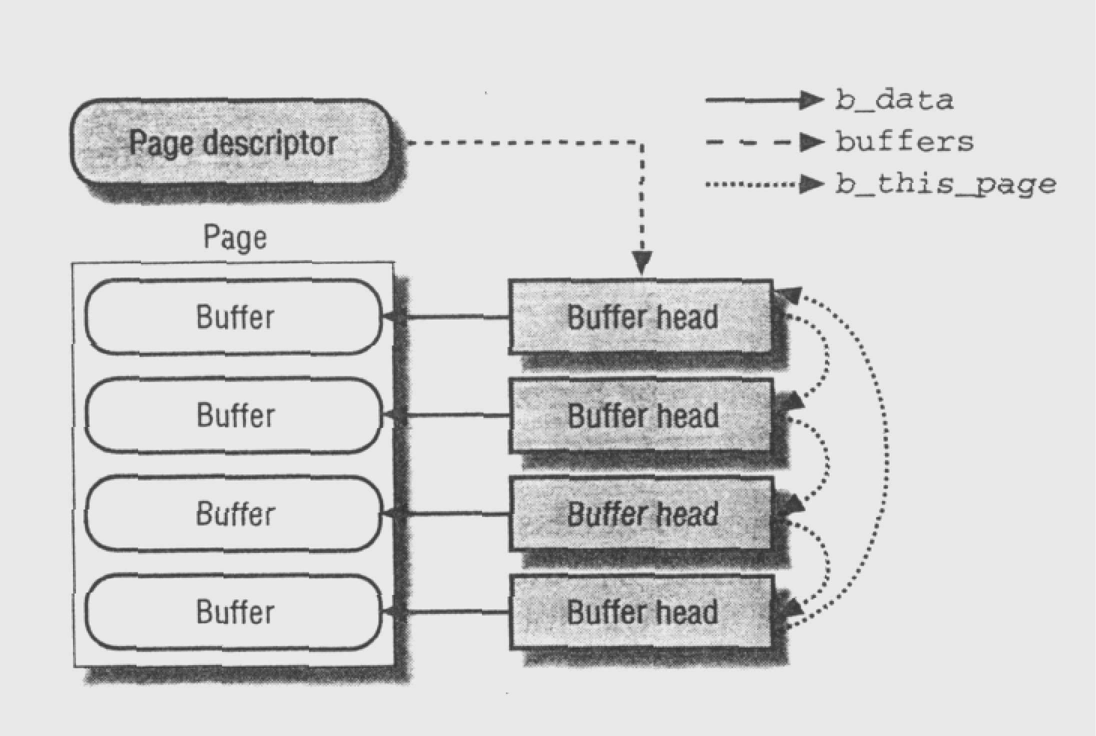
- buffer identified by device and block number
- `hash_table` helps to find buffer quickly
  - find_buffer()
  - insert_into_queues()
  - remove_from_queues()

getblk()

- main service routine for the buffer cache

Buffer Allocation

- not single memory objects for reasons of efficiency



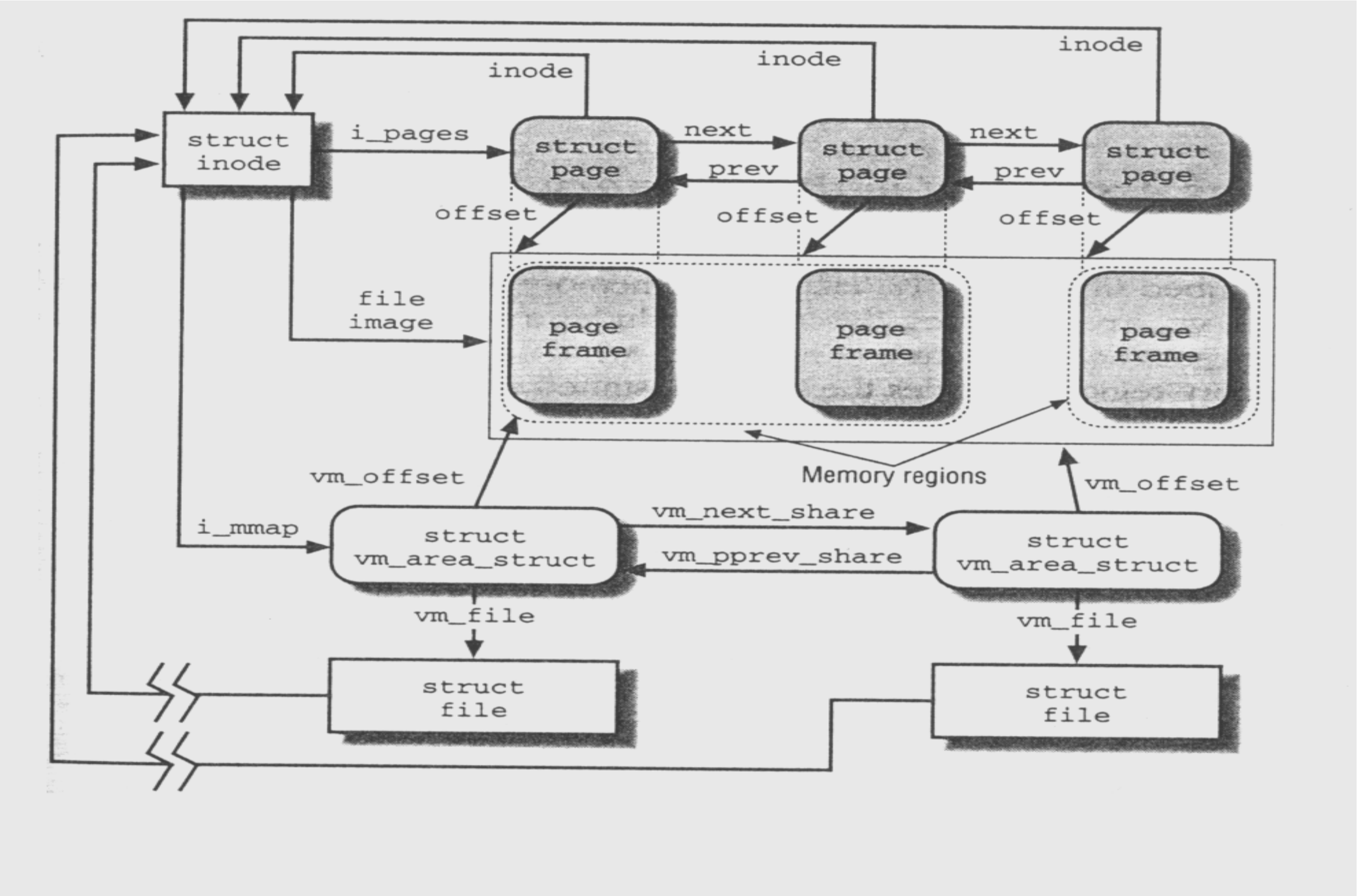
Page Cache

- all accesses through `read()`,`write()`, and `mmap()` are handled by the page cache
- blocks contained in page don't need to be adjacent on disk
  - device and block number not identifying
- file inode and offset are unique

Page Cache Data Structures

- page hash table
  - `struct page **page_hash_table;`
  - identified by inode and offset
  - size depends on memory available
- inode queue
  - all pages of an inode

Page Cache


inode
inode
inode
struct inode
i_pages
struct page
next
prev
struct page
next
prev
struct page
offset
offset
offset
file image
page frame
page frame
page frame
Memory regions
vm_offset
vm_offset
i_mmap
struct vm_area_struct
vm_next_share
vm_pprev_share
struct vm_area_struct
vm_file
vm_file
struct file
struct file