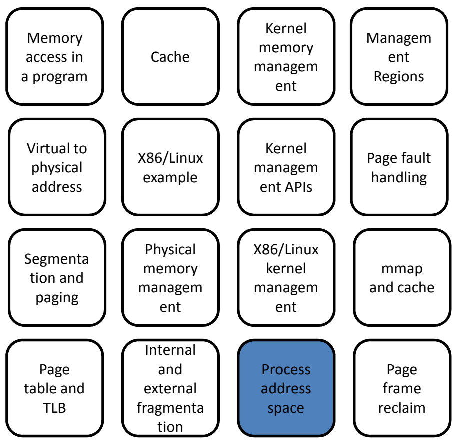
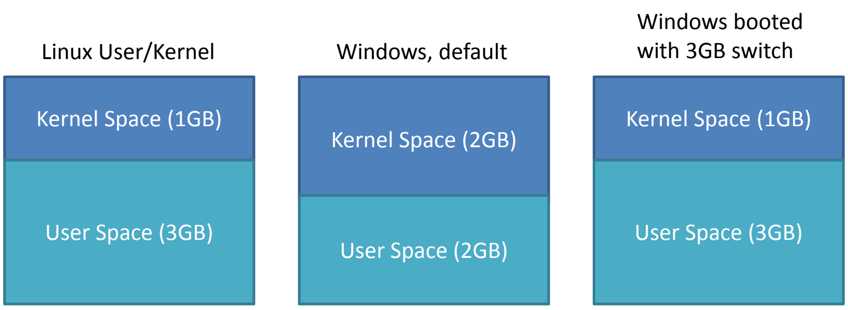
Operating System Design and Implementation

Memory Management – Part III

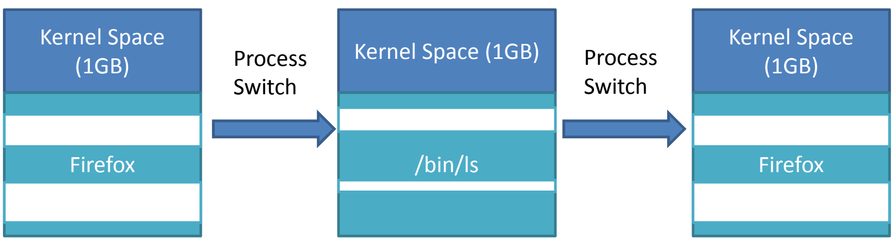
Shiao-Li Tsao



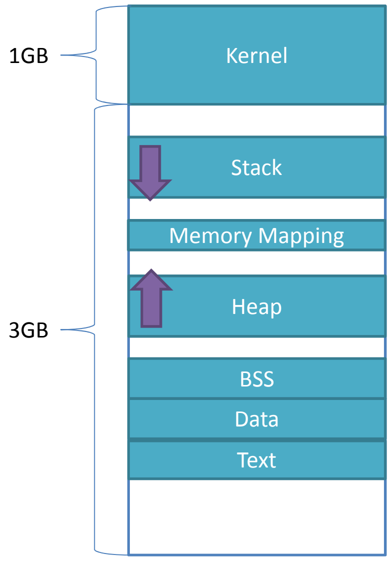
Kernel Space vs. User Space



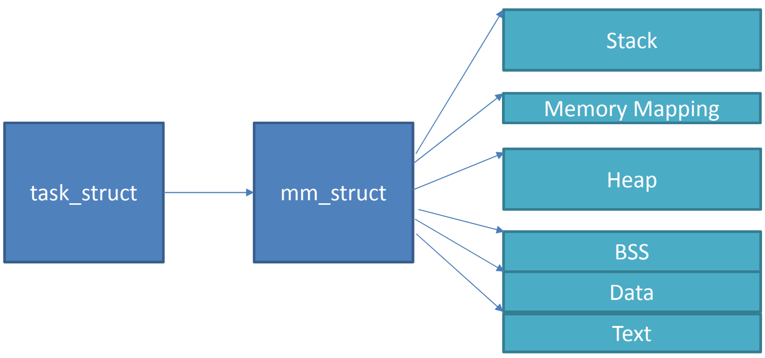
Virtual Addresses Before/After Context Switches

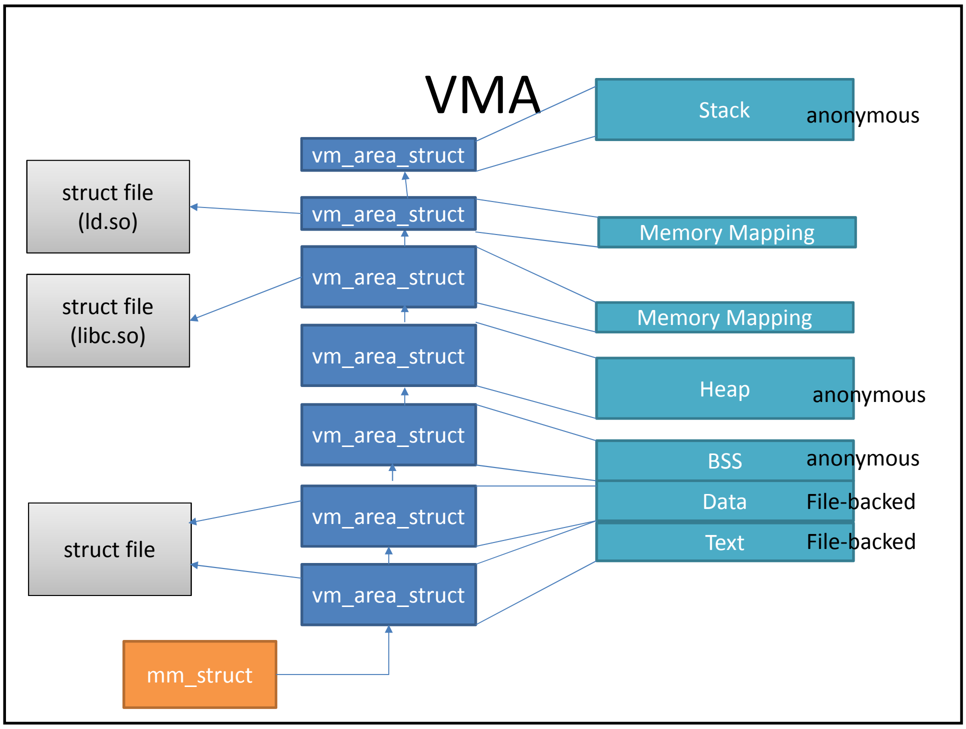


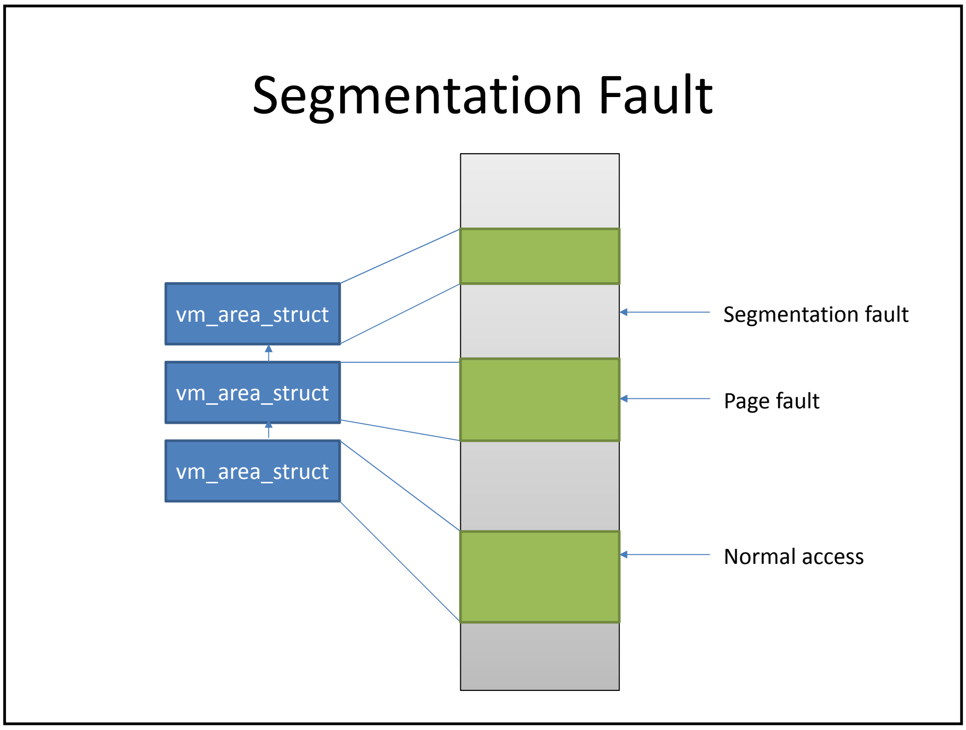
Per Process Address Space



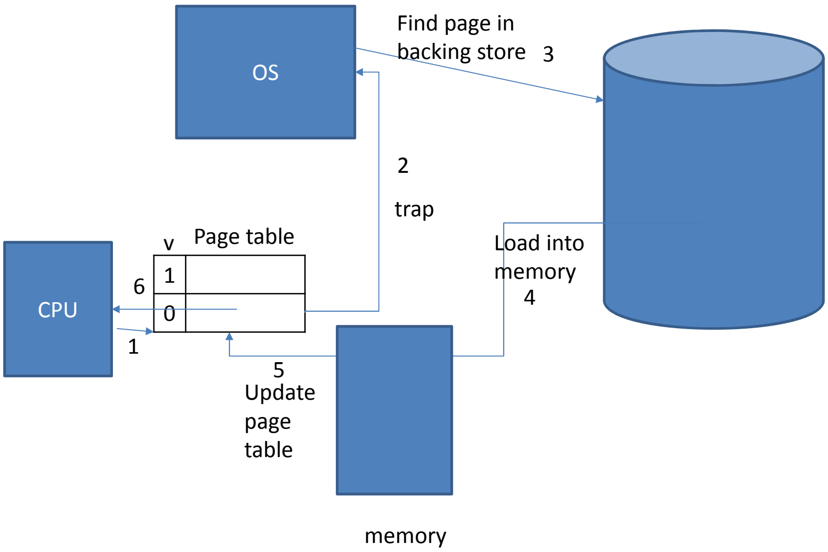
task_struct and mm_struct




VMA
vm_area_struct
vm_area_struct
vm_area_struct
vm_area_struct
vm_area_struct
vm_area_struct
vm_area_struct
struct file (ld.so)
struct file (libc.so)
struct file
mm_struct
Stack
anonymous
Memory Mapping
Memory Mapping
Heap
anonymous
BSS
anonymous
Data
File-backed
Text
File-backed


Segmentation Fault
vm_area_struct
vm_area_struct
vm_area_struct
Segmentation fault
Page fault
Normal access

Page Fault

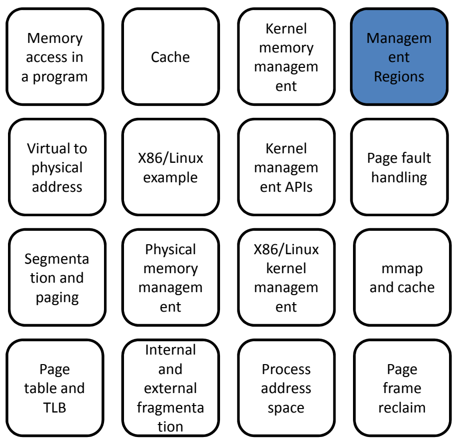


Allocation of MM Related Structures

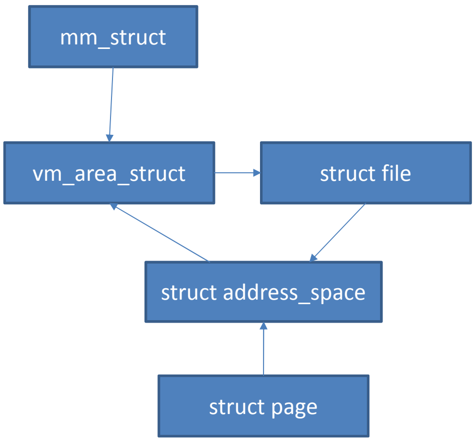
slabinfo - version: 1.1 (SMP)									
kmem_cache	80	80	248	5	5	1	:	252	126
urb_priv	0	0	64	0	0	1	:	252	126
tcp_bind_bucket	15	226	32	2	2	1	:	252	126
inode_cache	5714	5992	512	856	856	1	:	124	62
dentry_cache	5160	5160	128	172	172	1	:	252	126
mm_struct	240	240	160	10	10	1	:	252	126
vm_area_struct	3911	4480	96	112	112	1	:	252	126
size-64(DMA)	0	0	64	0	0	1	:	252	126
size-64	432	1357	64	23	23	1	:	252	126
size-32(DMA)	17	113	32	1	1	1	:	252	126
size-32	850	2712	32	24	24	1	:	252	126

Linux Kernel Thread and mm_struct

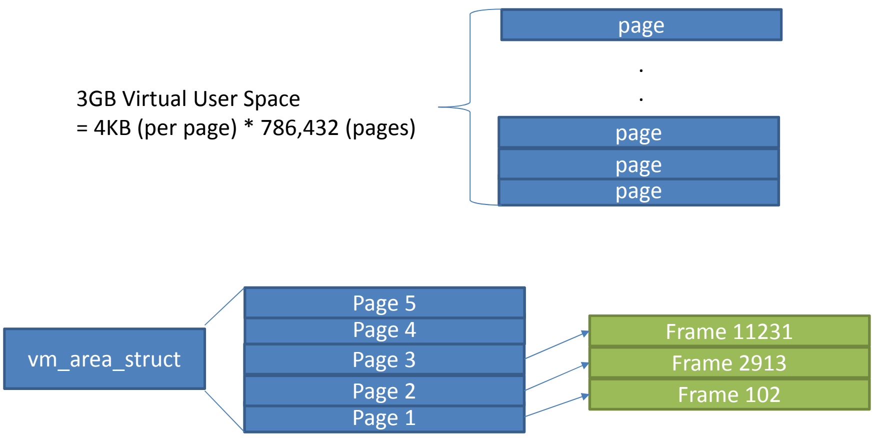
- Kernel threads do not have a process address space
  - mm field of a kernel thread's process descriptor is NULL
- Lack of an address space is fine, because kernel threads do not ever access any user-space memory
- Better performance



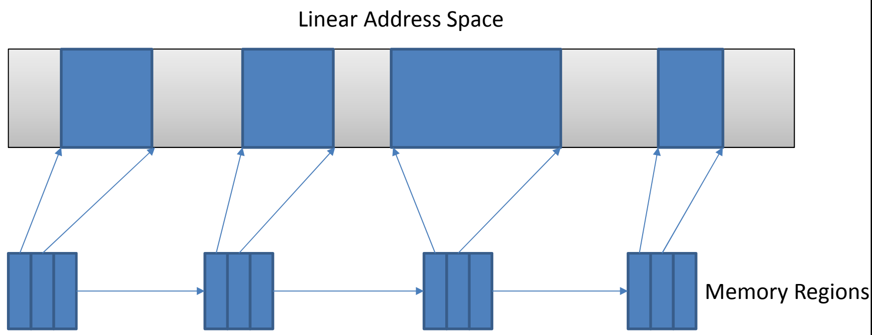
mm_struct, vm_area_struct, and page



VMA, Page and Frame

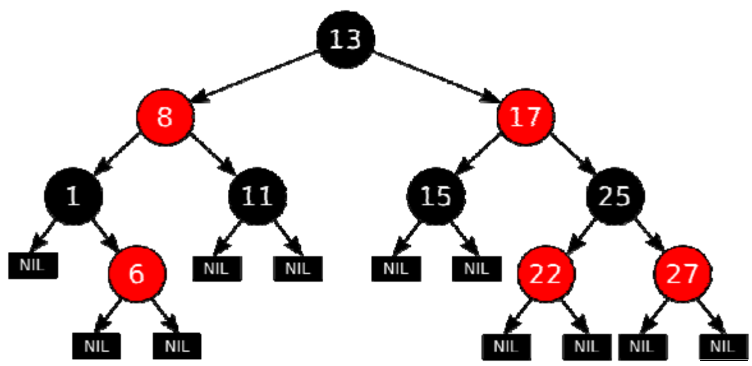


Memory Region

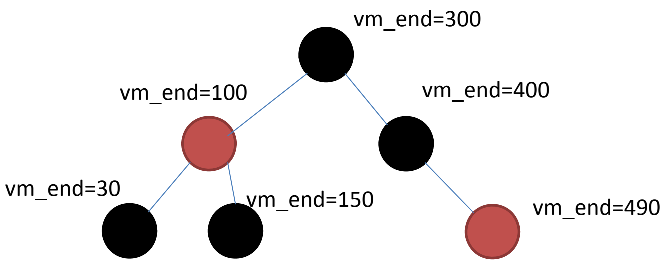


Red-Black Tree

- O(log n)



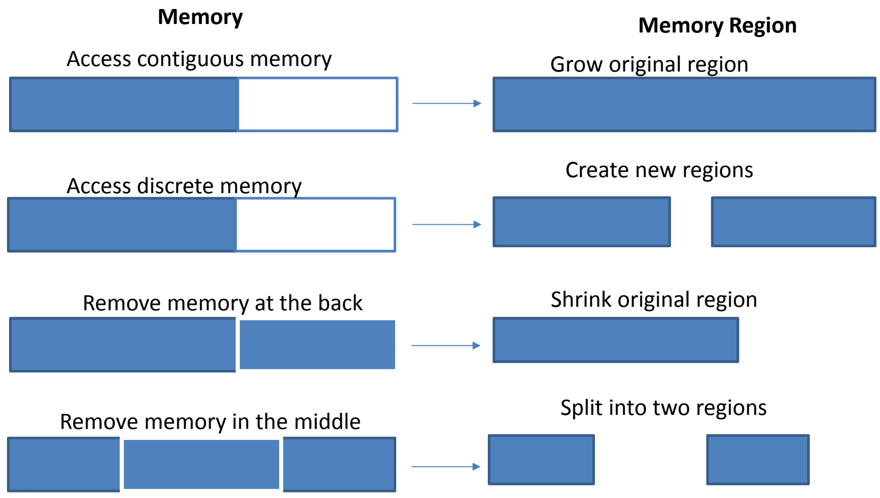
VMA Search Using R-B Tree



Ref: http://www.cs.columbia.edu/~junfeng/13fa-w4118/lectures/l20-adv-mm.pdf

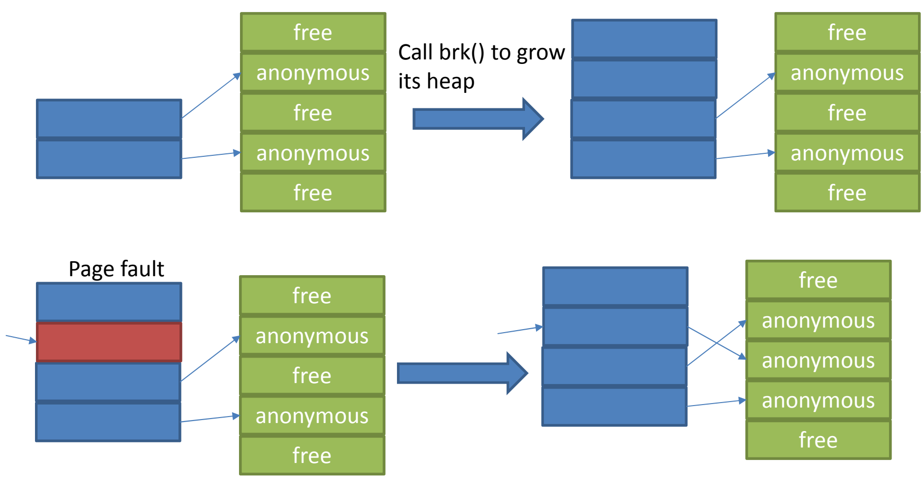
malloc() and Heap Management

Process Address Space

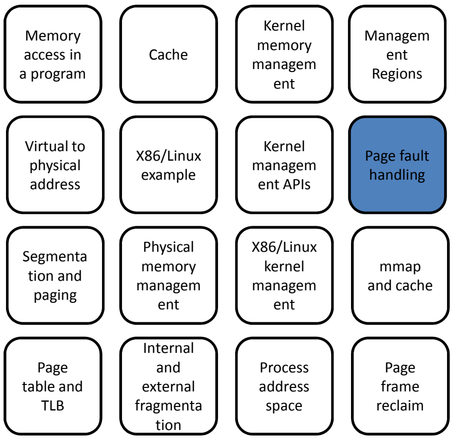


19

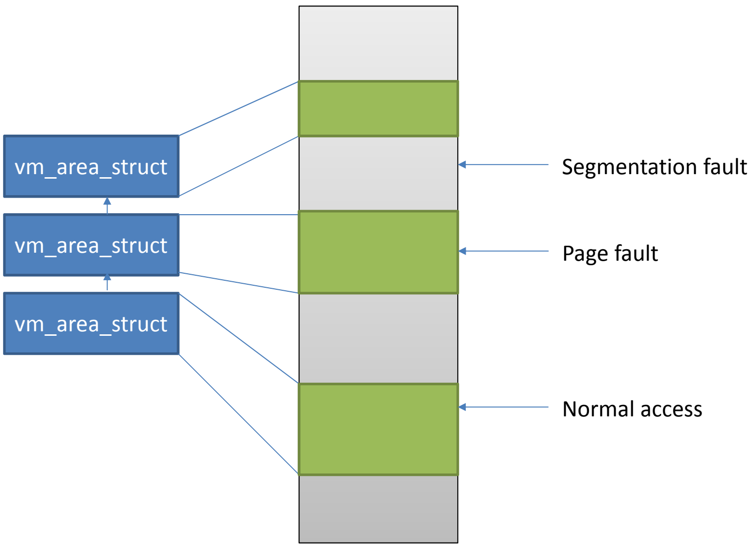
Enlarge VMA



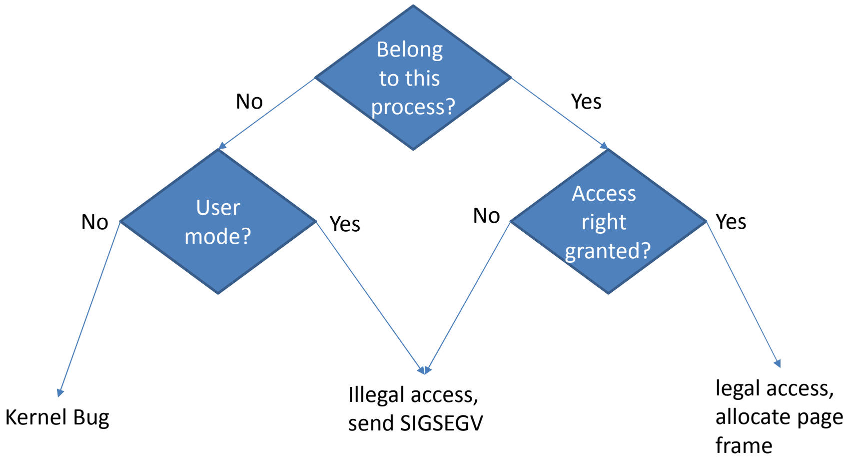
Ref: http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory/

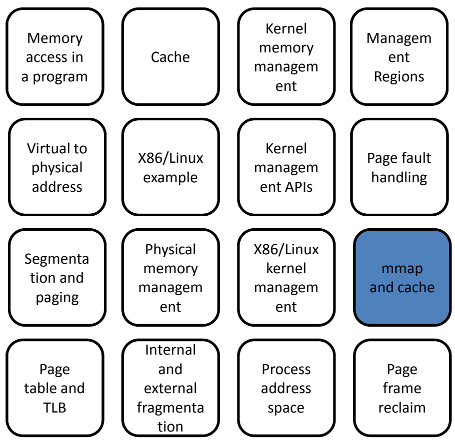


Segmentation Fault

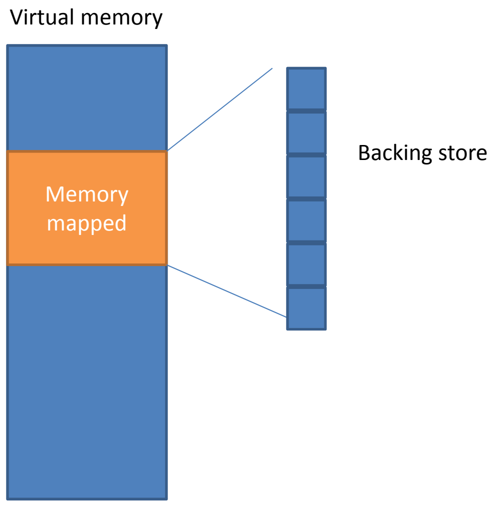


Page Fault Handler

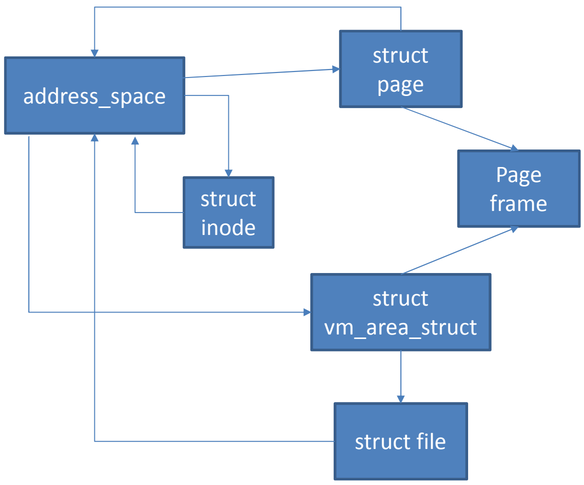




Memory Mapped



Data Structure for File Memory Mapping



Ref: Understanding the Linux Kernel, Daniel P. Bovet, Marco Cesati

Two Types of Memory Mapping

- A file mapping maps a memory region to a region of a file
  - backing store = file
  - as long as the mapping is established, the content of the file can be read from or written to using direct memory access (“as if they were variables”)
- An anonymous mappings maps a memory region to a fresh “virtual” memory area filled with 0
  - backing store = zero-ed memory area

Ref: Programmation Systèmes /Memory Mapping, Stefano Zacchiroli

Having memory mapped pages in common

- Thanks to virtual memory management, different processes can have mapped pages in common
- More precisely, mapped pages in different processes can refer to physical memory pages that have the same backing store
- That can happen in two ways:
  - through fork, as memory mappings are inherited by children
  - when multiple processes map the same region of a file

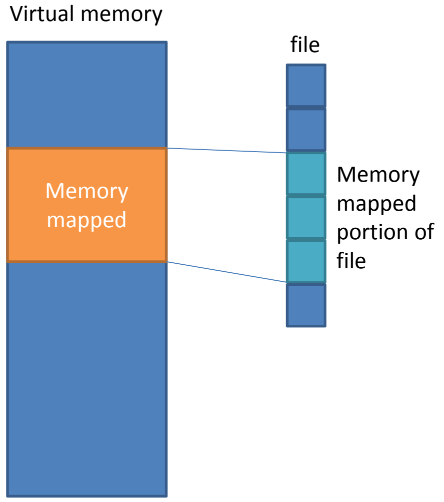
Ref: Programmation Systèmes /Memory Mapping, Stefano Zacchiroli

Shared vs private mappings

- With mapped pages in common, the involved processes might see changes performed by others to mapped pages in common, depending on whether the mapping is:
  - private mapping in this case modifications are not visible to other processes.
    - pages are initially the same, but modification are not shared, as it happens with copy-on-write memory after fork
    - private mappings are also known as copy-on-write mappings
  - shared mapping in this case modifications to mapped pages in common are visible to all involved processes
    - i.e. pages are not copied-on-write

Ref: Programmation Systèmes /Memory Mapping, Stefano Zacchiroli

File mapping and memory layout



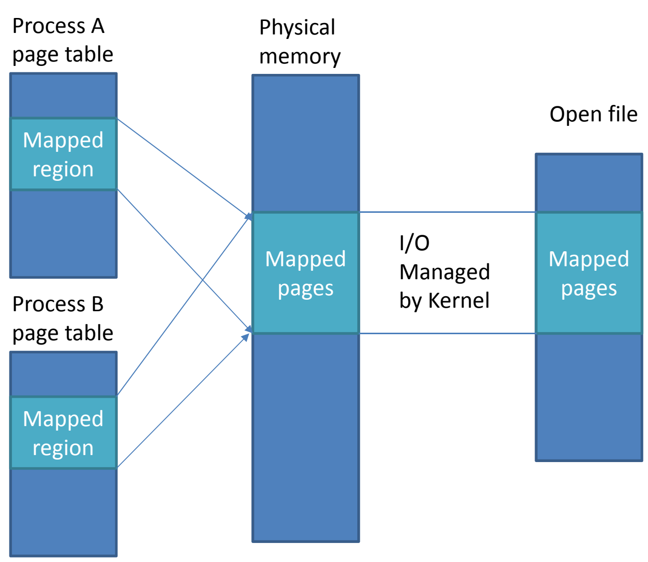
Ref: Programmation Systèmes /Memory Mapping, Stefano Zacchiroli

Shared file mapping (#2)

- Effects:
  - processes mapping the same region of a file share physical memory frames
    - more precisely: they have virtual memory pages that map to the same physical memory frames
  - additionally, the involved physical frames have the mapped file as ultimate backing store
    - i.e. modifications to the (shared) physical frames are saved to the mapped file on disk

Ref: Programmation Systèmes /Memory Mapping, Stefano Zacchiroli

Shared file mapping (#2)



Shared file mapping (#2) (cont.)

- Use cases
  - memory-mapped I/O, as an alternative to read/write
    - as in the case of private file mapping, but here it works for both reading and writing data
  - Inter-process communication, with the following characteristics:
    - data-transfer (not byte stream)
    - with filesystem persistence
    - among unrelated processes

Ref: Programmation Systèmes /Memory Mapping, Stefano Zacchiroli

Memory-mapped I/O

- Given that:
  - memory content is initialized from file
  - changes to memory are reflected to file
- we can perform I/O by simply changing bytes of memory.
- Access to file mappings is less intuitive than sequential read/write operations
  - the mental model is that of working on your data as a huge byte array (which is what memory is, after all)
  - a best practice to follow is that of defining struct-s that correspond to elements stored in the mapping, and copy them around with memcpy & co

Ref: Programmation Systèmes /Memory Mapping, Stefano Zacchiroli

Memory-mapped I/O — advantages

- performance gain: 1 memory copy
  - with read/write I/O each action involves 2 memory copies: 1 between user-space and kernel buffers + 1 between kernel
- buffers and the I/O device
  - with memory-mapped I/O only the 2nd copy remains
  - flash exercise: how many copies for standard I/O?
- performance gain: no context switch
  - no syscall and no context switch is involved in accessing mapped memory
  - page faults are possible, though reduced memory usage
  - we avoid user-space buffers ! less memory needed
  - if memory mapped region is shared, we use only one set of
- buffers for all processes seeking is simplified
  - no need of explicit lseek, just pointer manipulation

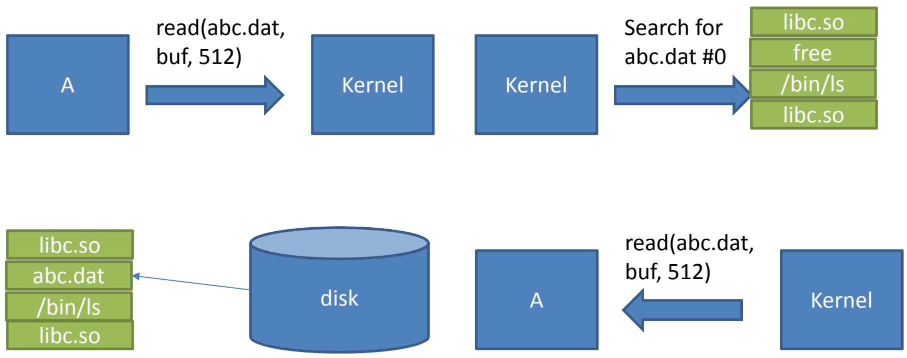
Ref: Programmation Systèmes /Memory Mapping, Stefano Zacchiroli

Memory-mapped I/O — disadvantages

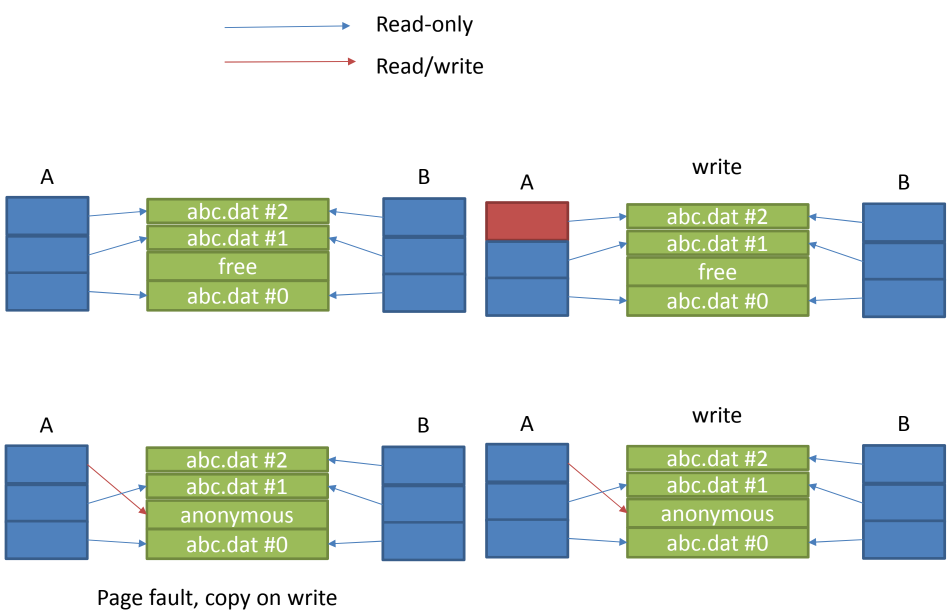
- memory garbage
  - the size of mapped regions is a multiple of system page size
  - mapping regions which are way smaller than that can result in a
- significant waste of memory
- memory mapping must fit in the process address space
  - on 32 bits systems, a large number of mappings of various sizes might result in memory fragmentation
  - it then becomes harder to find continuous space to grant large memory mappings
  - the problem is substantially diminished on 64 bits systems
- there is kernel overhead in maintaining mappings
  - for small mappings, the overhead can dominate the advantages
  - memory mapped I/O is best used with large files and random access

Ref: Programmation Systèmes /Memory Mapping, Stefano Zacchiroli

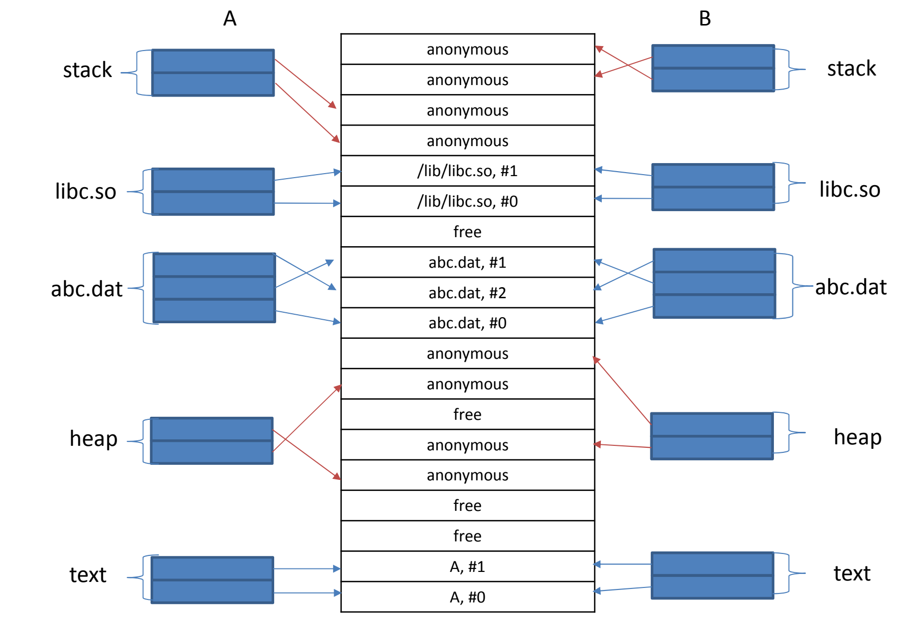
Page Cache



Ref: http://duartes.org/gustavo/blog/post/page-cache-the-affair-between-memory-and-files/



Ref: http://duartes.org/gustavo/blog/post/page-cache-the-affair-between-memory-and-files/



Ref: http://duartes.org/gustavo/blog/post/page-cache-the-affair-between-memory-and-files/

Memory access in a program	Cache	Kernel memory managem ent	Managem ent Regions
Virtual to physical address	X86/Linux example	Kernel managem ent APIs	Page fault handling
Segmenta tion and paging	Physical memory managem ent	X86/Linux kernel managem ent	mmap and cache
Page table and TLB	Internal and external fragmenta tion	Process address space	**Page frame reclaim**

The Types of Pages Considered by the PFRA

- ***Ref: Understanding the Linux Kernel, Daniel P. Bovet, Marco Cesati***

- ***Ref: CS161: Operating Systems, Matt Welsh***

Implementation of PFRA

- ***Ref: Understanding the Linux Kernel, Daniel P. Bovet, Marco Cesati***