VM

Prof. K.Gopinath

A primary function of an OS: manage its memory; MMU
CPU speed: mem speed: disk speed = 1: 5 : ???

OS very simple if no mem mgmt:
- one program at a time in memory, loaded contiguously,
- simple linking/loading, no addr translation
- RT/embedded or very small systems

<u>Requirements:</u>
- run a program larger than phys memory (avoid overlays)
- prog need not know about phys mem config (HW independence)
- allocate & manage mem resources
- run partially loaded programs (reduce startup time)
- run >1 program at same time (incr CPU util)
- movable programs during execution (relocatable)
- allow sharing (share text; DLLs)

Monoprogramming without swapping or paging

Multiprogramming and memory usage
- Utilization of CPU with n processes with prob p for I/O wait: 1-p^n
- Fixed partitions (example: IBM OS/360 MFT)
- Relocation and protection (base & limit registers)

Swapping: interactive systems; swap space on disk
- Multiprogramming with variable partitions
  - holes, compaction?, stack/data segment growth
  - Mem mgmt with bit maps/linked lists/buddy system; coalescing
  - first/best/next/quick fit with linked lists/bit maps
  - internal and external fragmentation with buddy system

- 50% rule: if # of processes in memory is n, mean number holes n/2.
  - In equilibrium: half of the ops above allocs and the other half deallocs; on avg, one hole for 2 procs
- unused memory rule: (n/2)*k*s=m-n*s
  - s, k*s: avg size of process, hole; m: total memory
  - fraction of memory wasted: k/k+2
- overhead of paging: process size*size of page entry/page size
  - +pagezise/2

some interesting results

- if n is the number of allocated areas, then n/2 is the number of holes for “simple” alloc algs (not buddy!) in equilibrium
- dynamic storage alloc strategies that never relocate reserved blocks: mem eff not guaranteed!
  - with blocks 1 and 2, can run out of mem even with 2/3rds full
    - 23 seats in a row, groups of 1 and 2 arrive; do we need to split any pair to seat? no more than 16 present
      - Solution: no single is given seat 2, 5, 8, … 20
    - not possible with 22 seats; no more than 14 present

VM: AS as an abstraction to free program from phys mem locs

- CPU gen VA; trans to real addr
- HW+SW have to cooperate
- costly: about 10% CPU time on busy sys+ fragmentation (paging)

OLD solutions:

- AS size smaller than phys mem
  - PDP-11; 16bit AS=> 64KB I+ 64KB D
  - also curr SPARC/IA-32: 32bit AS size but 36 bit phys mem
  - also curr RS/6000: each process has 32 bit AS size but 52 bit system VA size
- overlay: needs info on phys mem config
- curr out-of-core computations
- swapping: few in memory at same time and swapped out to swap (disk) if nec

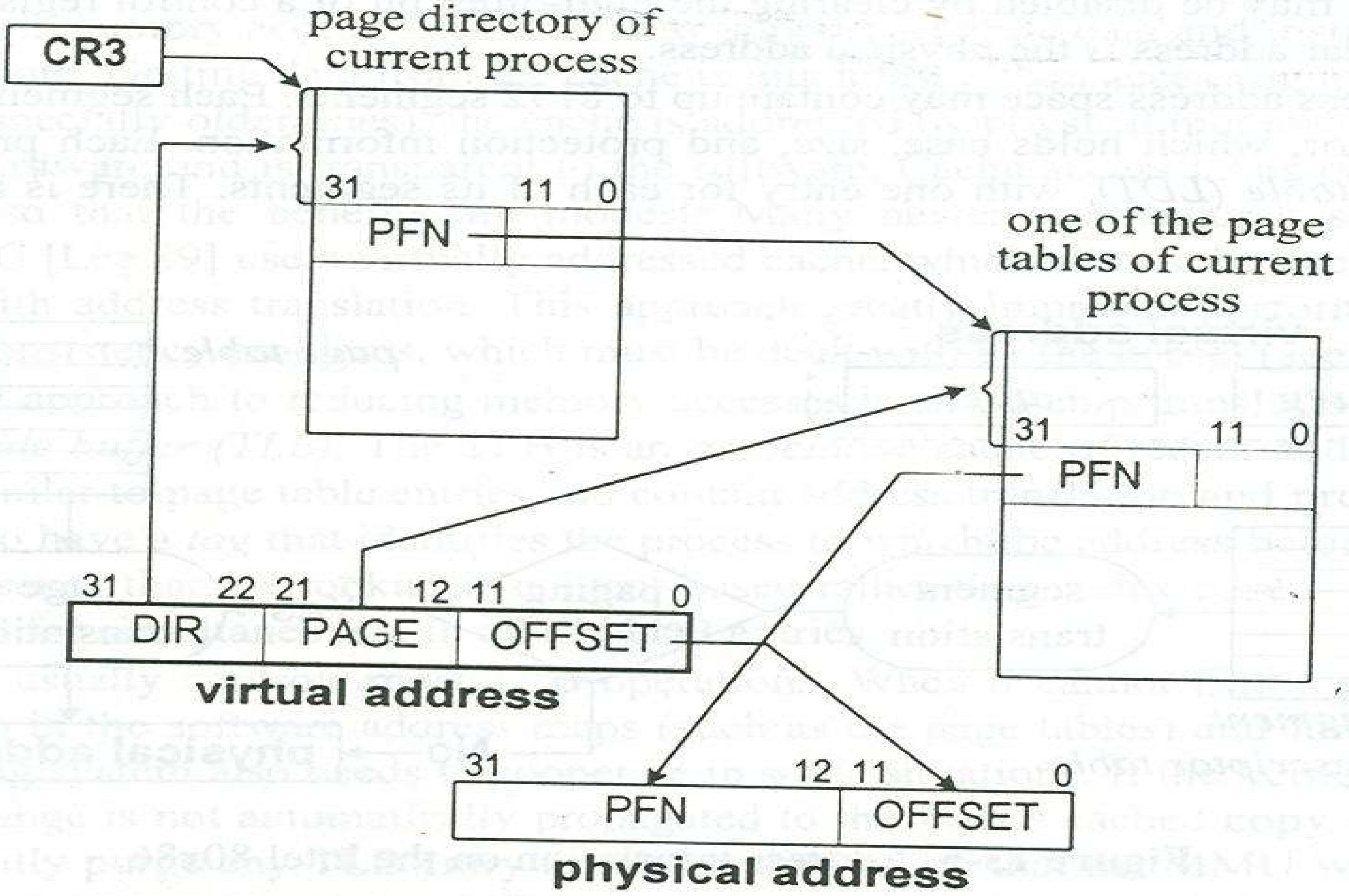
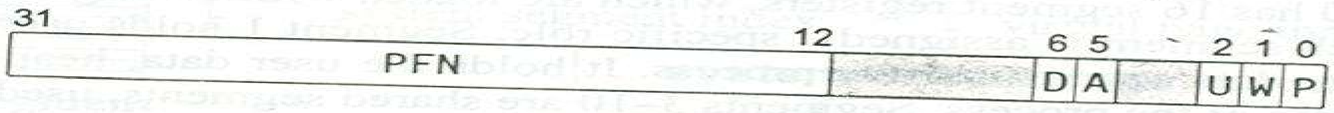


Figure 13-6. Address translation on Intel *x86*.



PFN	Page Frame Number
D	Dirty
A	Accessed (Referenced)
U	User (0) / Supervisor (1)
W	Read (0) / Write (1)
P	Present (valid)

Figure 13-7. Intel *x86* page table entry.

“NEW” solution:

Demand paging: page frame (phys page), virtual page; RWX perms on page

- page fault, fault handler
- page: unit of mem alloc, prot, addr trans
  - min faulted in: arch-dep: MVC (6B) on IBM 370: 2 pages for inst itself; the block of chars to move can straddle 2 pages each
- page into mem only on ref; also possible: anticipatory paging
- adv: large AS, fast program startup, multiprogramming, better than swapping
- critical: trans from VA to PA has to be efficient

Segmentation: segments contiguous, base + len, all accesses checked to be within

Segment+Paging: segment need not be phys mem contiguous

Logical Segmentation: on Unix, high-level abstraction of contiguous virt pages

Functional requirements of a demand-paged arch:

addr space mgmt: fork, exec, vfork, exit, change size of data/stack segment, add a new region (shm)

addr trans: maps

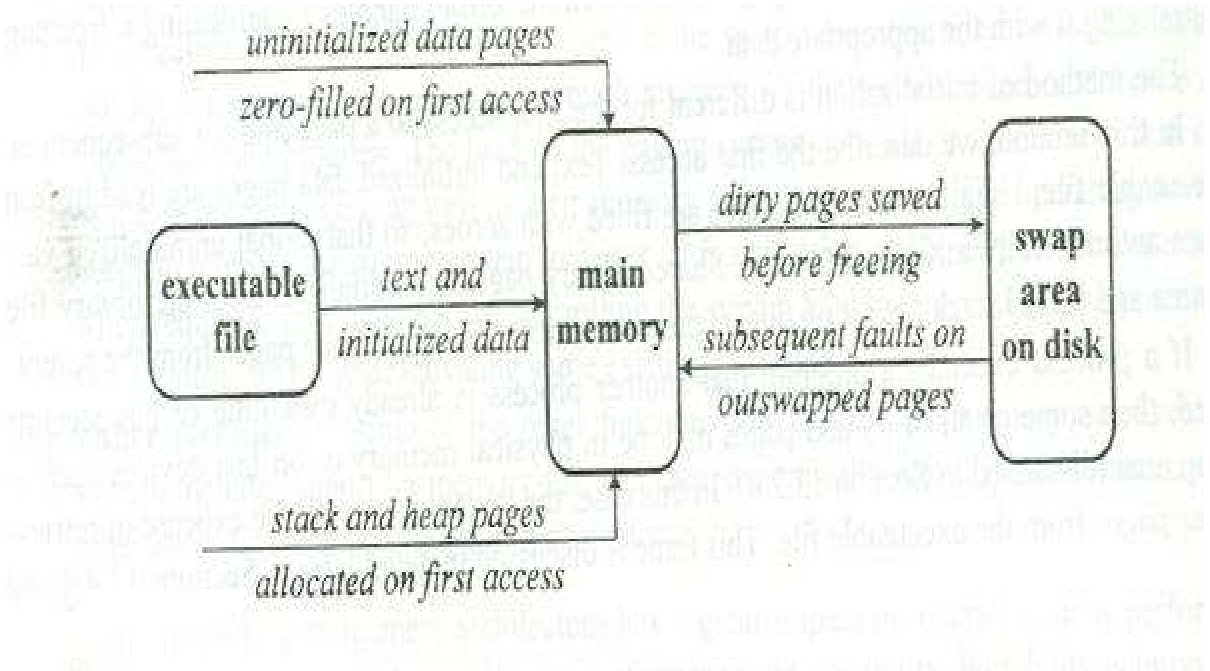
phys mem mgmt: kernel/users; phys memory as cache: consistency/currency

mem prot: kernel from users, one user from another, user from himself (code/data), SIGSEGV

mem sharing: text, COW in fork/exec, DLL, monitor system load: avoid thrashing, ...

VAS: differences in protection, init, sharing for diff types:

- text
- initialized data (read from file; possibly from swap: sticky!)
- uninitialized data (zfod)
- stack, heap (alloc on first access, later from swap)
- modified data (from swap)
- shared memory (prot set on first alloc)
- shared libraries (text shared but data COW)



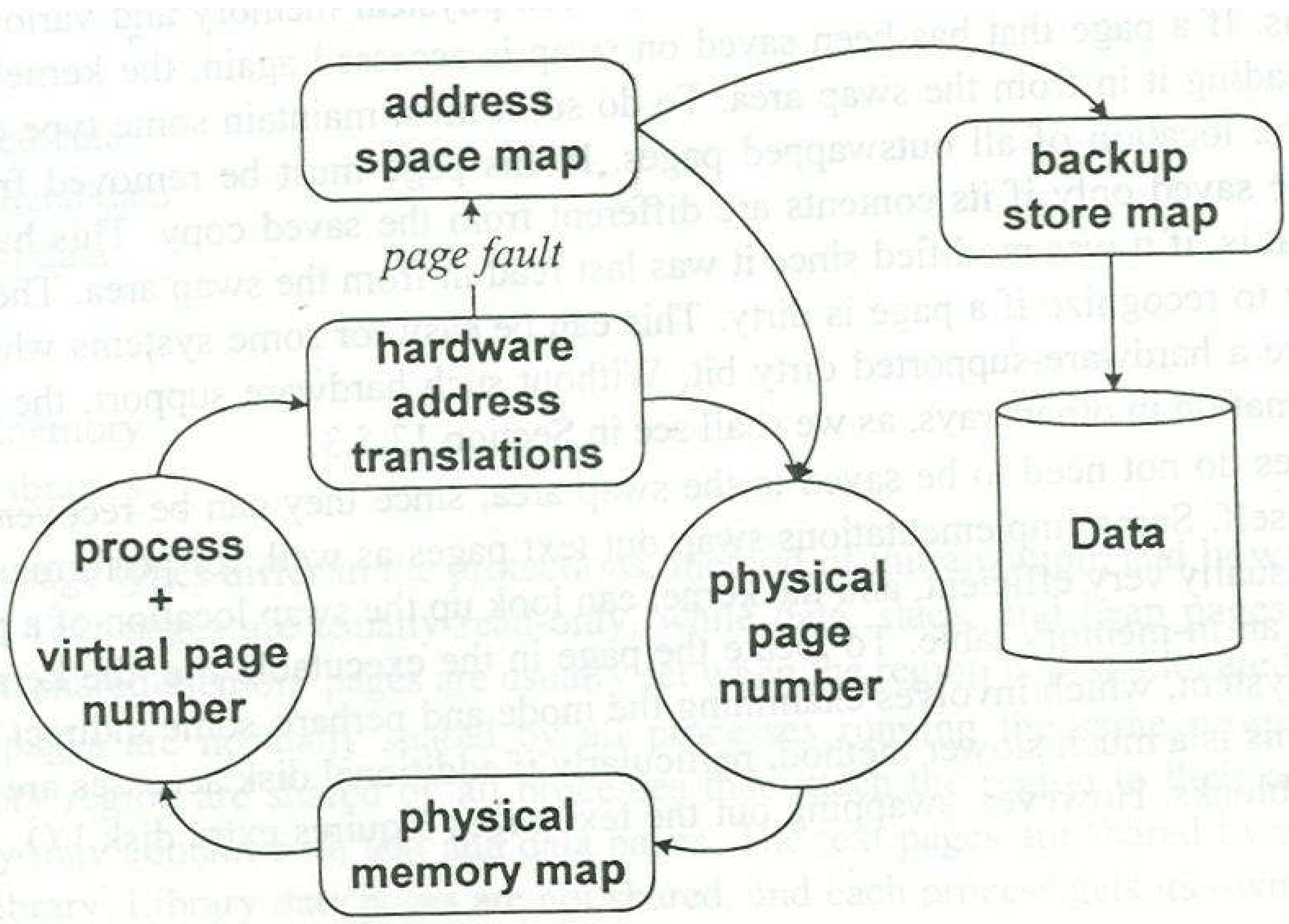


Figure 13-4. Address translations.

Translation maps:

HW addr translations: TLB, ...

Addr Space Map: full maps; HW(TLB) may have only small (MIPS) or inexact (sim: no referenced bit in VAX/R3000)

Phys mem map: reverse mapping to remove trans for a virt page on a phys page

Backing store map: file?, shared lib obj file?, swap device

Page Repl Policies: working set, locality, LRU

local: working set; Unixware

global: common with min number for each process

allocation algs: equal, proportional to size of process, priority. .

free of a page: reclaim only when necessary or (better) maintain a pool of free pages

control thrashing by working set model or page fault freq, prepaging, analyse program structure (array algs)

I/O interlock, locking in mem (high priority, real-time, ...)

Probs of circular dependencies with native root/swap mirroring

- One plex has to be detached on failure
- need txn (say, 2-phase) to effect detachment
- volume has to be locked as volume config changing and system concurrent
- config daemon a user process but its page may be on swap
- if 2-phase transaction (daemon & kernel), deadlock!

Possible Solutions

- plock daemon (expensive: 5MB ker mem)
- “single-phase” kernel txn wihout daemon
- recognize swapper process specially
- write special code for root/swap mirroring

Simlarly: mem alloc problems:

  initiating I/O requires a small amt of mem

  swapper fails to get mem to do I/O to swap & sleeps: DEADLOCK!

SVR4 VM Arch:

concept of file mapping central

- user mmap: lets users map file/AS to AS/file and use mem insts for R/W
- kernel map: entire AS seen as mappings to diff objs (files, ...)
  - HP-UX: has user mmap + std buf cache;
  - AIX 3.1: only in kernel (no mmap)
  - SVR4, AIX4.1: both

SVR4 integrates 3 diff ways of accessing files for achieving consistency: demand paging of executables, mmap'ed files, read/write to open files

- last method reads data into buffer cache in old Unix
- "new" Unix: map files into kernel VAS <vnode, offset>
- old buf cache mostly redundant except for metadata (superblocks, indir blocks, inodes, directories) as these not repr as <vnode, offset>: uses <dev, blk #>

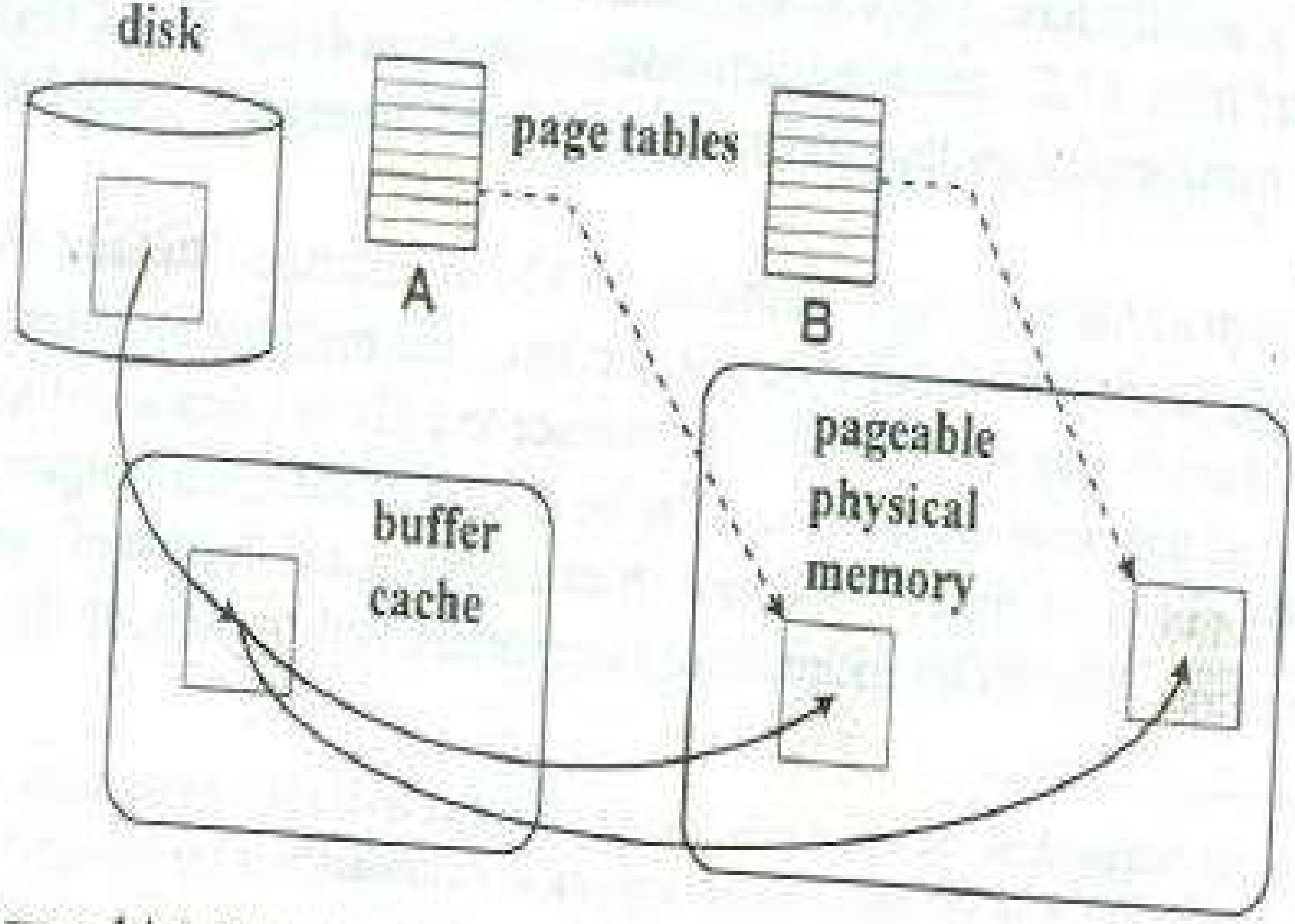


Figure 14-1. Two processes read the same page in traditional UNIX.

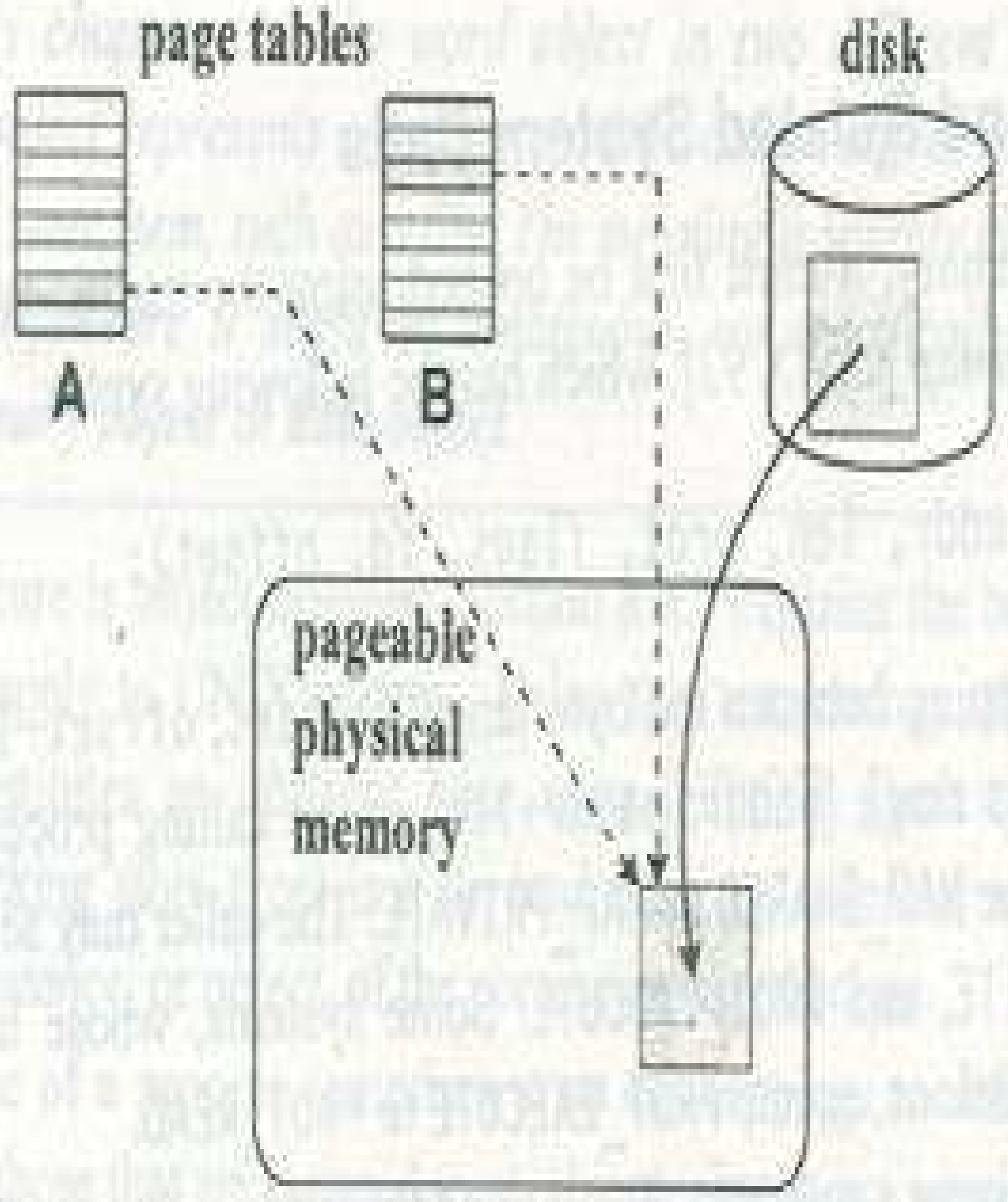


Figure 14-2. Two processes map the same page into their address space

read/write atomic whereas mmap atomic only at word/byte level

sharing of a file by 2 processes A&B using rdwri: 1 in buffer cache, 1 in each AS (1 mem2mem copy into each AS)

sharing of a file by 2 processes A&B using mmap: 1 in mem shared

Two types of sharing: shared and private

- Private: on write, COW.

paddr = mmap (addr, len, prot, flags, fd, offset)

- flags: MAP_SHARED, MAP_PRIVATE, MAP_FIXED
- prot: PROT_READ, PROT_WRITE, PROT_EXECUTE

munmap (addr, len) ; mprotect (addr, len, prot) mem obj: abstraction of mapping between a region of mem and backing store

- several types of backing store: swap space, local/remote files, frame bufs
- unify all these types thru OO
- AS: a set of mappings to different data objs
- fs provides name space for mem objs
- vnode layer: VM subsystem interacts with FS
  - many mem objs can be mapped to a vnode but a mem obj associated with only 1 vnode
  - if mem obj not associated with a file: anonymous obj: user stack, etc.
- VM arch indep of Unix: hence non-Unix sys can use VM; hat for portability
- physical mem server as a cache of mem objs; uses COW extensively

struct page, struct as, struct seg, struct hat, struct anon

```
                           anon layer->swap layer=>swap device
                           ^     struct proc
                           ^       v
FS<=vnode layer<- as layer=>VAS
         v                              v
    page layer=>phys mem<-hat layer       (Fig 14.3/14.5)
```

page: each page mapped into by 1+ mem objs but only 1 backing store (vnode); identity given by <vnode, offset>

struct has vnode ptr, offset in vnode, hash chain ptrs (hashed on id), ptrs for vnode page list, ptrs for being on free list or I/O list (to disk), flags (locked, wanted, in-transit), refcount for sharing thru COW, hat-info (copies of mod & ref bits, all translations of a page)

lowlevel i/f for finding a page given identity, add/del from hash Qs & synch

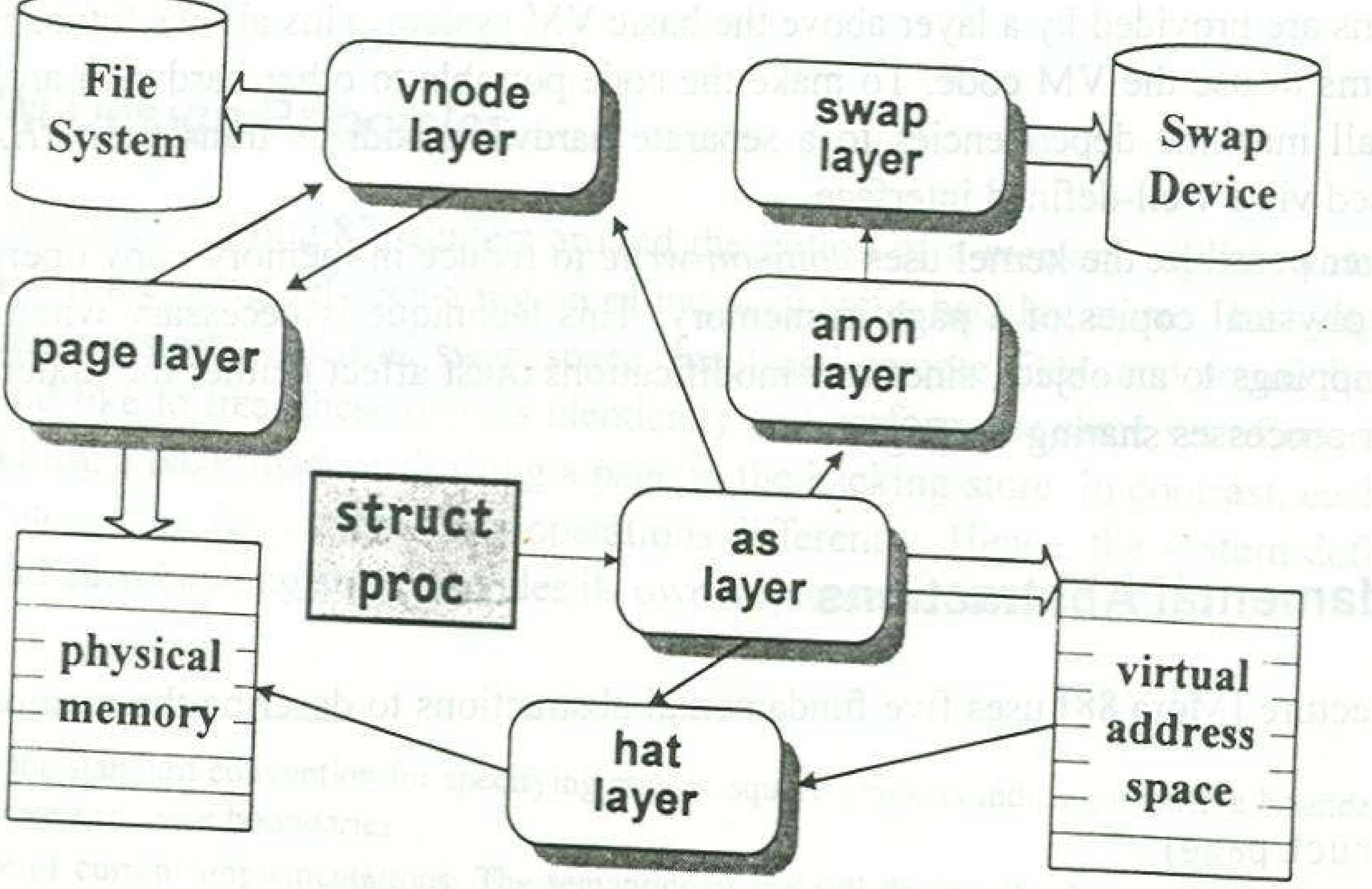


Figure 14-3. The SVR4 VM architecture.

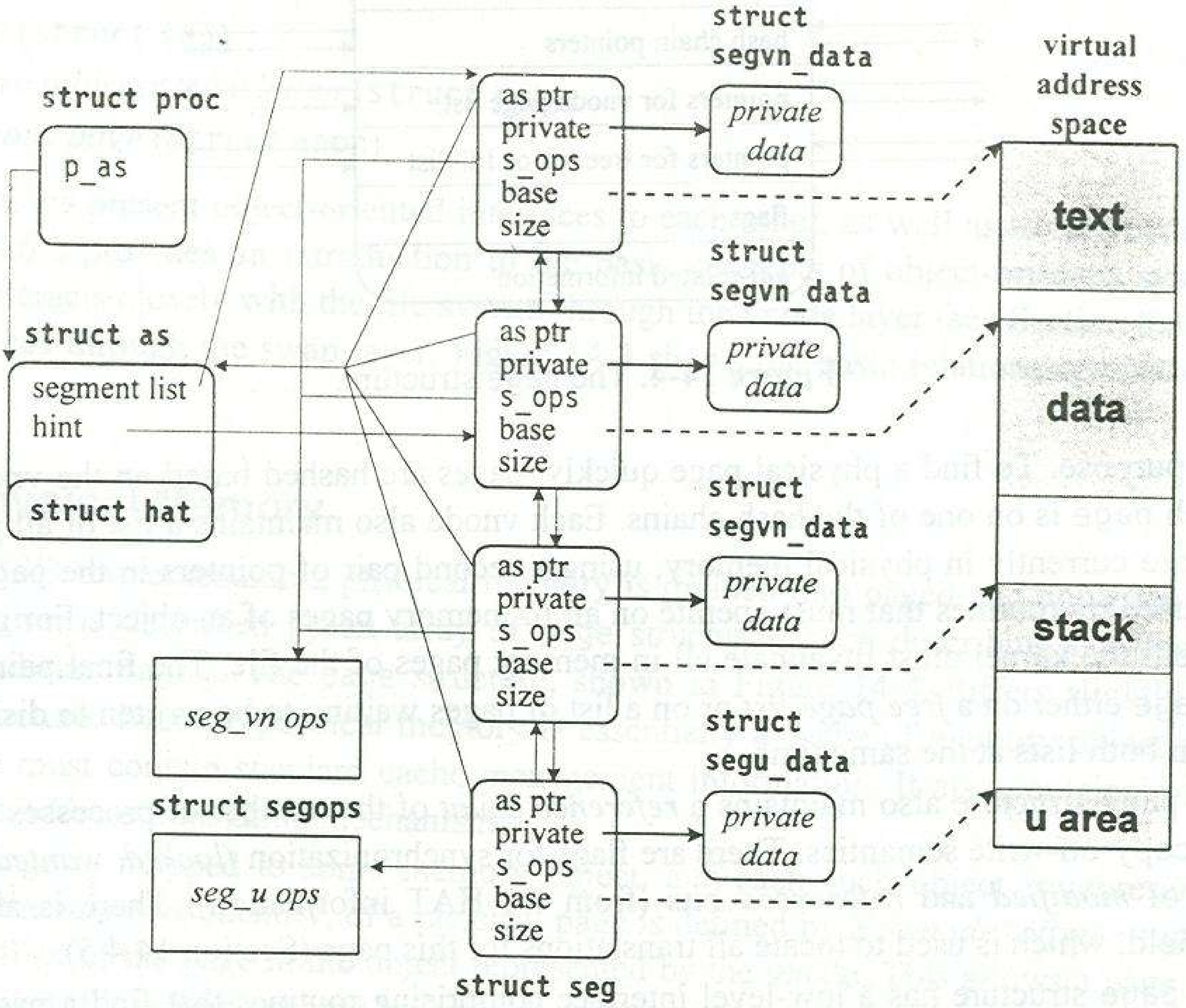


Figure 14-5. Describing the process address space.

as: primary per-process abstraction; provides high-level i/f to process AS; struct proc points to as

struct as has hdr of linked list of segs (non-overlapping, shared/priv, page-aligned addr ranges sorted by base addr), struct hat, AS size, hint to the last seg that faulted, synch flags, resident set

as layer: as_alloc (fork/exec), as_free (exec/exit), as_dup (fork), as_map, as_unmap (map/unmap mem obj into as; mmap/munmap/...), as_setprot, as_checkprot (mprotect), as_fault (handler), as_faulta (faultahead)

seg: identical oo i/f to rest of VM (Fig 14.7)

base class: base, size, ptr to as, fwd/bwd ptrs on seglist

virt funcs: seg_ops (dup, fault/faulta, set/chkprot, unmap, sync, swapout)

private data: s_data

addl virt func seg_create (needed as called before seg_ops initialized)

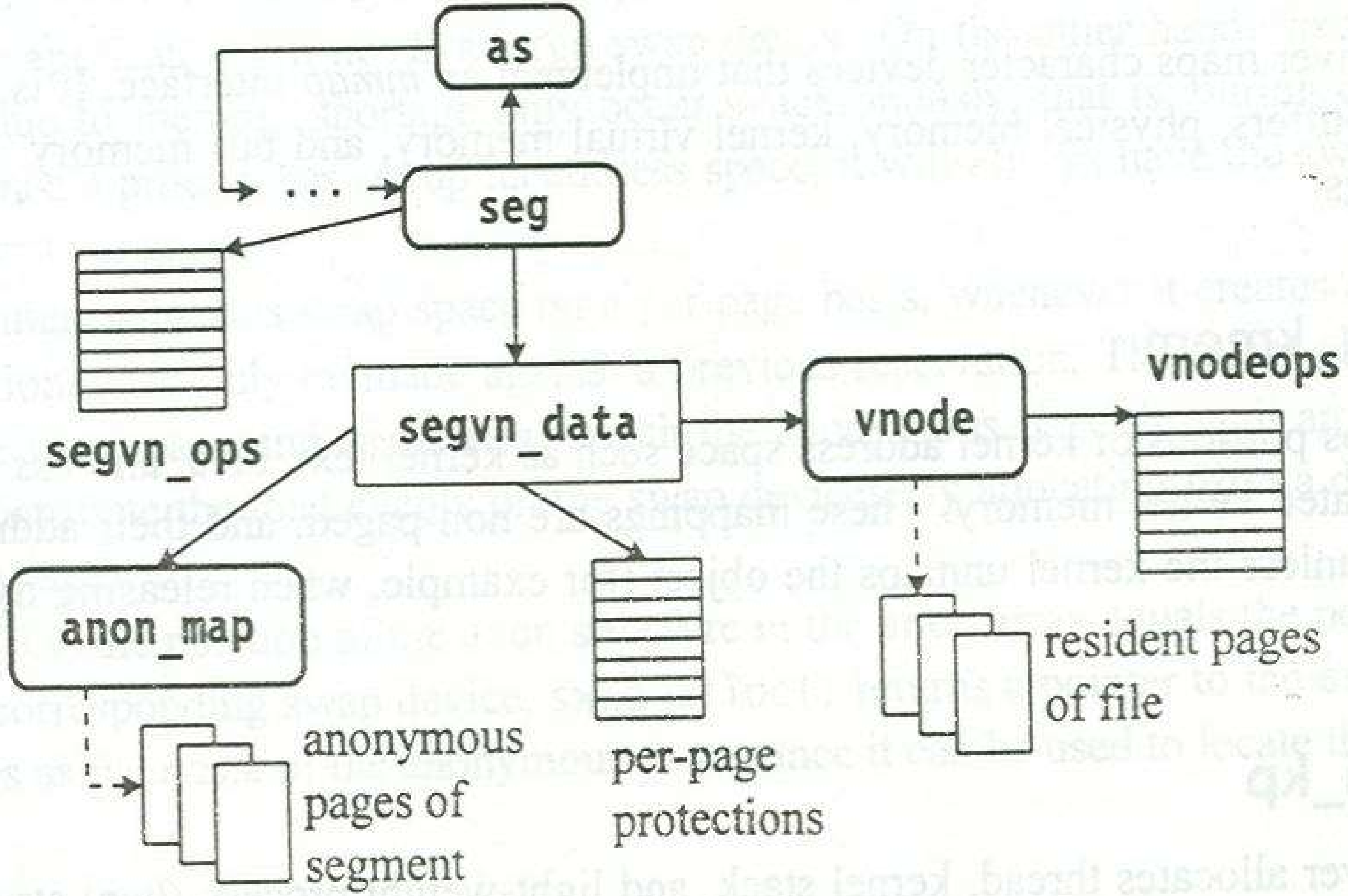


Figure 14-7. Data structures associated with a vnode segment.

anon page: created when a process modifies a MAP_PRIVATE mapping to an obj

- has no perm storage; discarded on exit of process or unmap
- initialized pages mapped to file but become anon after 1st mod
    - ref count for sharing
    - swap layer provides backing store

anon obj: source of all zeroes (/dev/zero)

- uninitialized data/stack: MAP_PRIVATE mapping to anon obj
- on first access, these data/stack regions become anon pages
    - but anon obj (/dev/zero) DOES NOT provide backing store!
    - Swap does...

anon interface: anon_dup (dupl refs to a set of anon pages; incr ref counts), anon_free, anon_private (makes a priv copy of a page & alloc new anon), anon_zero (creates a 0-filled page with a new anon), anon_getpage (resolves a fault, read from swap-device if nec) hat: all hw-dep code (HW addr translation) but redundant info!

Info can be discarded at will and rebuilt from machine-indep layer

setup & maintain mappings reqd by MMU (page tables, translation buffers)

sole i/f between kernel and MMU but opaque to VM subsytem

ops on hat_layer: hat_alloc, hat_free, hat_dup (dupl translations during fork), hat_swapin, hat_swapout(to rebuild/release hat info on swap in/out)

ops on a range of pages of a process: hat_chgprot, hat_unload (inval xlations & flush corresp TLB entries), hat_memload (load xlation for 1 page), hat_devload (used by seg_dev)

ops on all xlations of a given page: hat_pageunload (unload all xlations: inval PTE & flush TLB entry), hat_pagesync (upd mod/ref bits in all xlations from struct page) (Fig 14.6)

keeps all xlations of a shared page on a linked list and stores the list ptr in hat-dep field in struct page

ref port of SVR4 to Intel uses mapping chunk to monitor all xlations for a given page

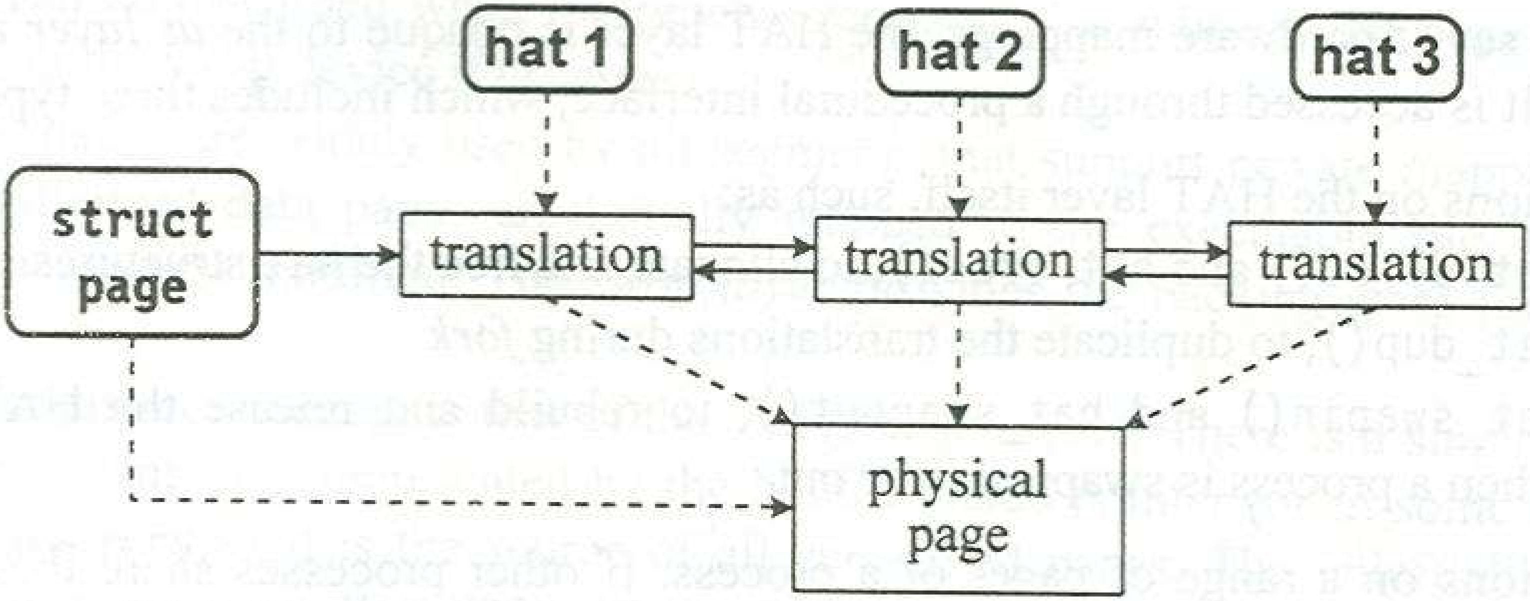


Figure 14-6. Locating all translations to a page.

<u>seg drivers</u>:

- seg_vn: mappings to reg files and to the anon obj
- seg_map: kernel internal mappings to regular files
- seg_dev: mappings to char devices that impl mmap (frame bufs, phys mem, kernel virt mem, bus mem etc.)
- seg_kmem: misc nonpaged mappings for kernel text, data, bss & dyn alloc kernel mem
- seg_u: mappings to u area
- seg_objs: map kernel objs to user space
- seg_kp: for MT systems:
  - allocates thread, kernel stack, LWP structs
  - may be in pageable or non- mem
  - also allocates red zones (single w-prot page at end of stack) to prevent kernel stack ovfl

seg_vn: maps user addr to reg files (executable text & initialized) or anon obj (bss, user stack)

addl thru shared mem or mmap

has per-segment curr & max prot (set on init map) or per-page prot array

ptr to vnode of mapped file (vnode provides all ops on file)

mapping type (shared or private), offset in file,

anon_map:ptr to anon pages: mod pages of private mappings

seg_map: optimized version of seg_vn providing quick but transitory kernel mappings to files

solves "old" consistency problem with fs: buf cache has one copy of disk block & mmap possibly elsewhere

segmap_getmap (maps part of a vnode to KVA)

segmap_release (release the map & upd disk if mod)

Fig 14.13/14

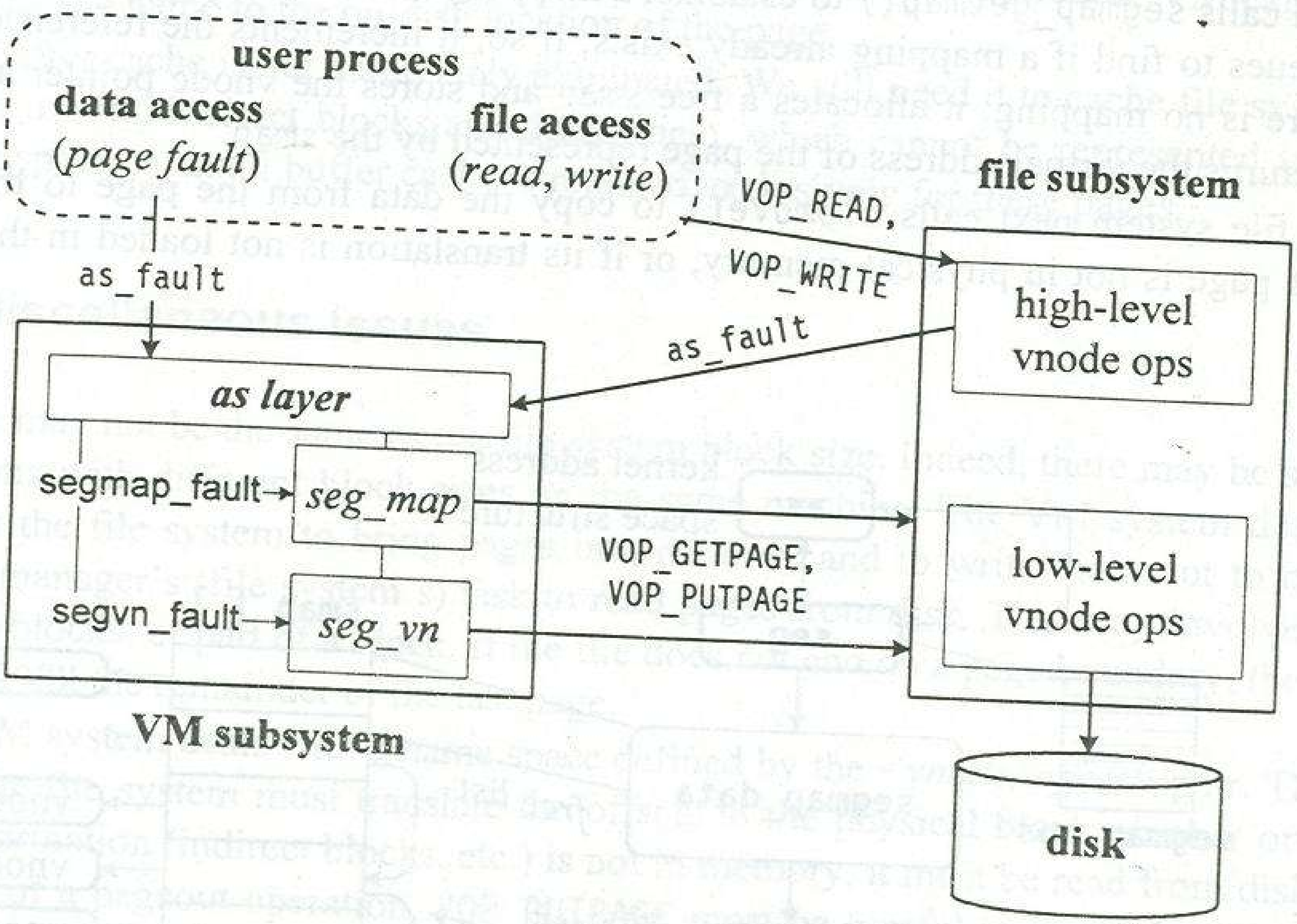


Figure 14-13. Relationship between file and VM subsystems.

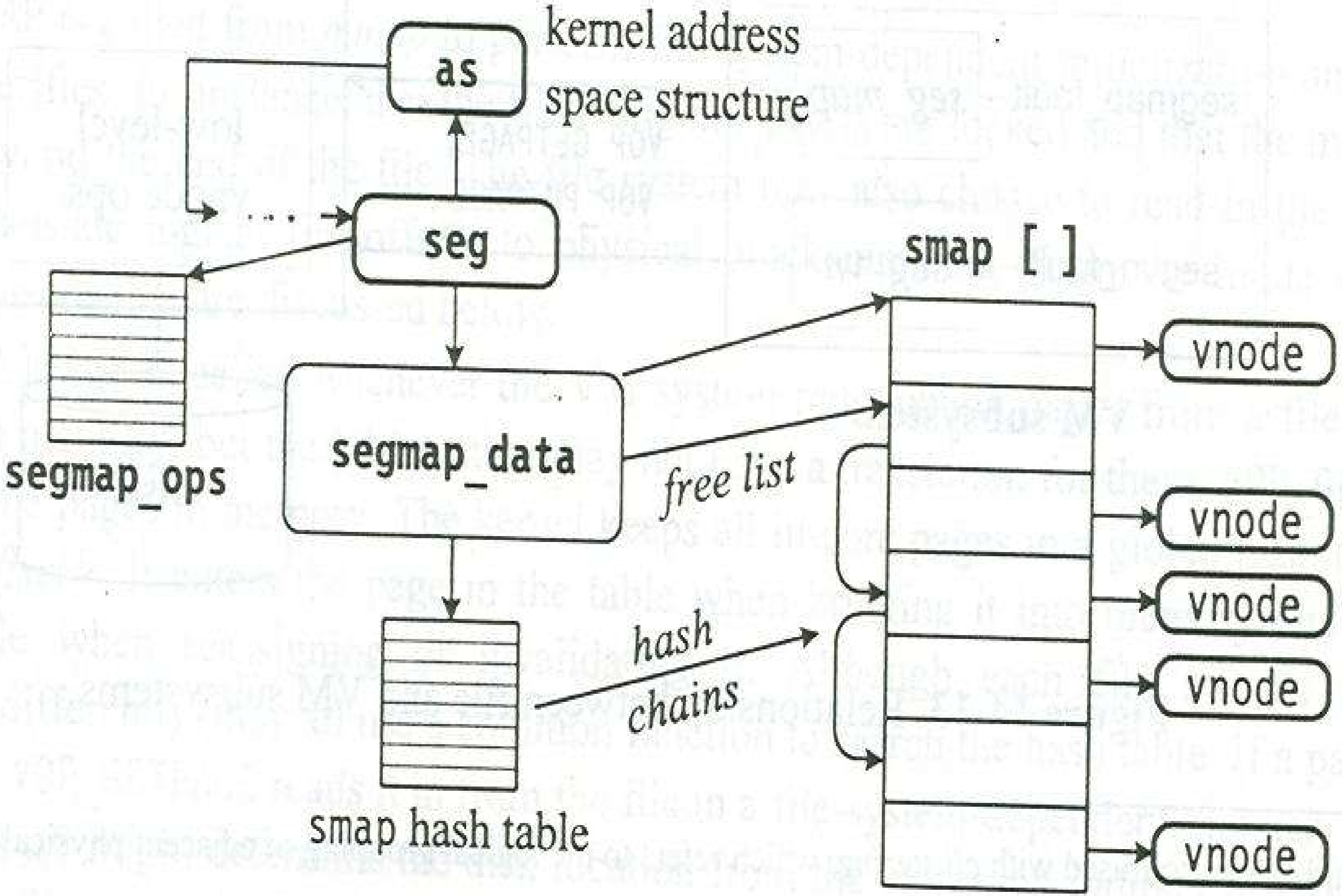


Figure 14-14. seg_map data structures.

SWAP layer: maintains information necessary to locate page

- page in mem: anon struct stores ptr to page struct
- page not in mem: swap_xlate (anon struct) ret vnode/offset in swap using VOP_GETPAGE
- segments must both reserve and allocate swap space; conservative policy
- per-page allocs of swap (swap_alloc/_free): fragmentation of swap dev
- struct anon []: one element for each page on dev
- swapinfo for each swap dev: vn/offset, ptrs to beg/end of anon array
  - maintinains free list and on list of swapinfos for devs

Fig 14.8/9

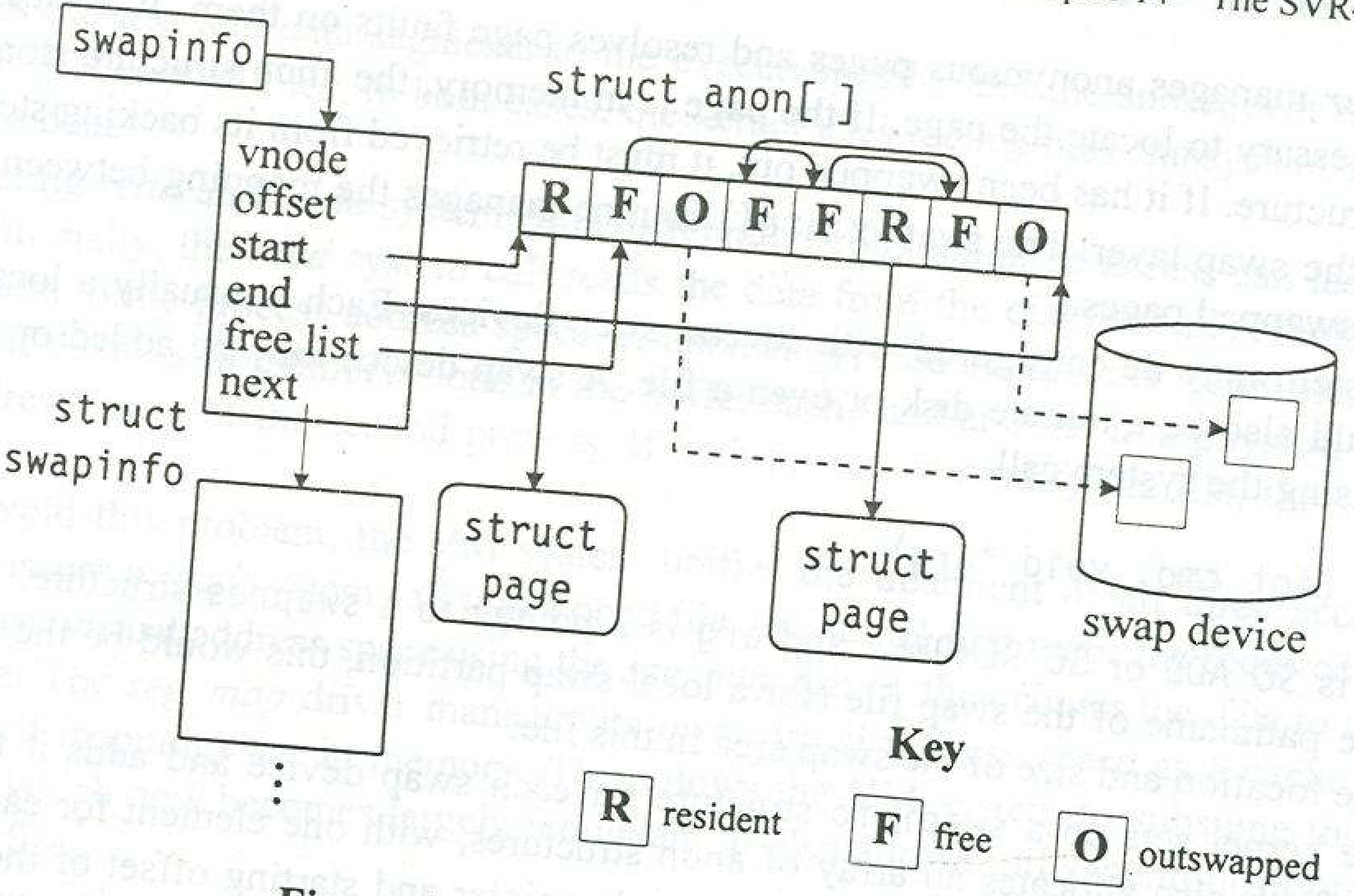


Figure 14-8. Swap layer data structures

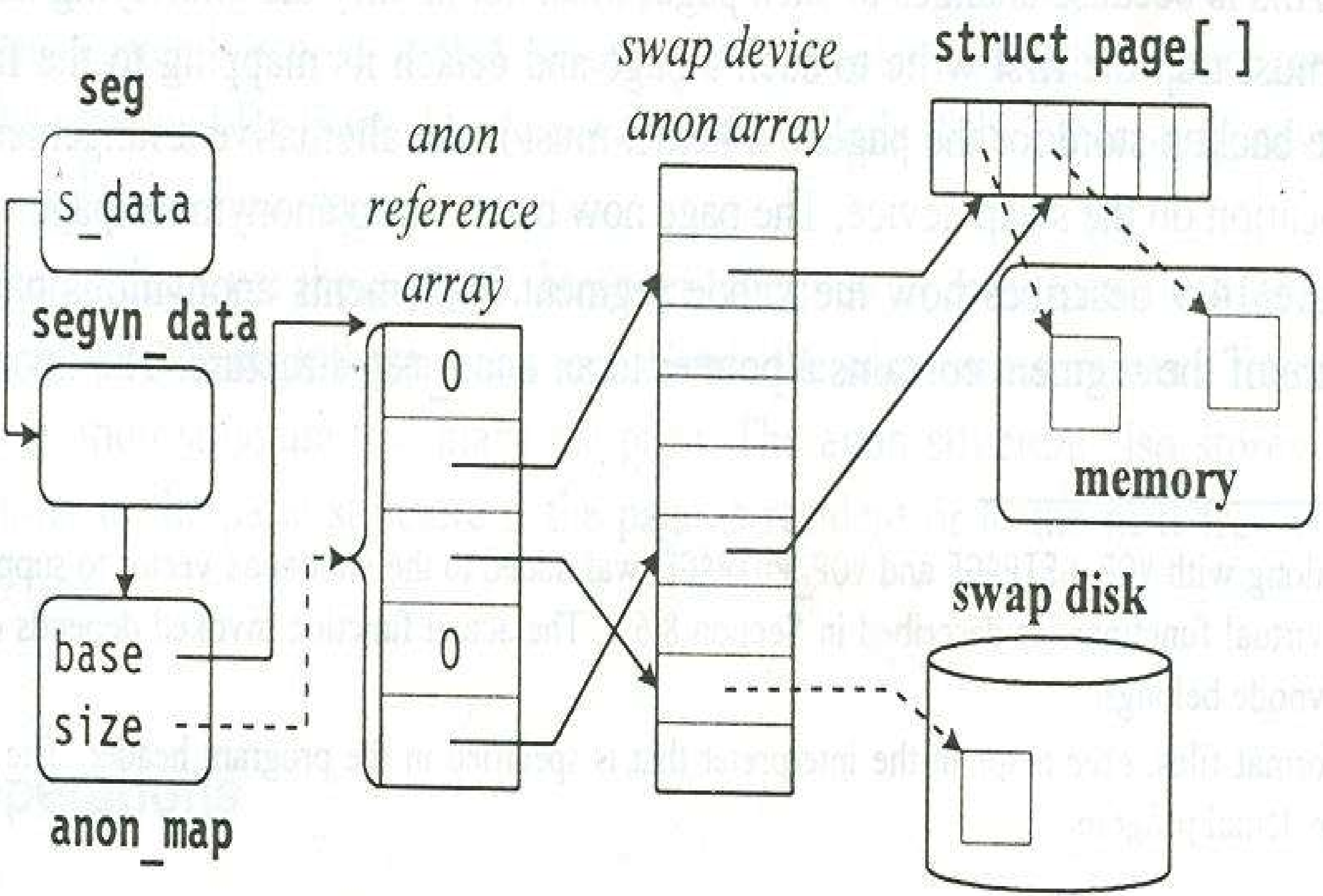


Figure 14-9. Anonymous pages of a vnode segment.

SVR4: pos of a anon struct in anon array==pos of swap page in swap dev

- swap_alloc returns ptr to anon struct: "name" of anon page
- swapctl(int cmd, void *argument)
    - cmd: SC_ADD/_REMOVE/_LIST/...
    - argument: ptr to swapres struct
- segments that have anon mem have anon ref array: one element for each page in segment
    - if page non-anon, then NULL
- anon struct: ref count, ptr to page (if resident) or next free elem if free
    - ptr to new anon (if one swap dev removed and its swap realloc to another swap device)

VM ops:

Creating a new mapping: exec or mmap

- VOP_MAP called: checks if another mapping exists and deletes it (as_unmap)
- as_map then called: allocs a seg & create on it
- mmap: checks that perms !> opened with; seg records maxprot; mprotect checks maxprot
- exec: private mappings to text, data, stack regions
    - text: mapped PROT_READ/_EXECUTE to file; initialized:PROT_WRITE to file; bss & stack: anon obj

as_free to free AS (exit)

anon page handling: anon pages created by

- write to MAP_PRIVATE page mapped to file/anon obj
- 1st access to shared mem page

swap has to be alloc on creation as backing store gone

private part of seg has ptr to anon_map which points to anon ref array

- for each page of seg, array element NULL if page not anonymous or refers to one anon struct (pointing to page/swap)

Fig 14.10/11

Fig 14.12

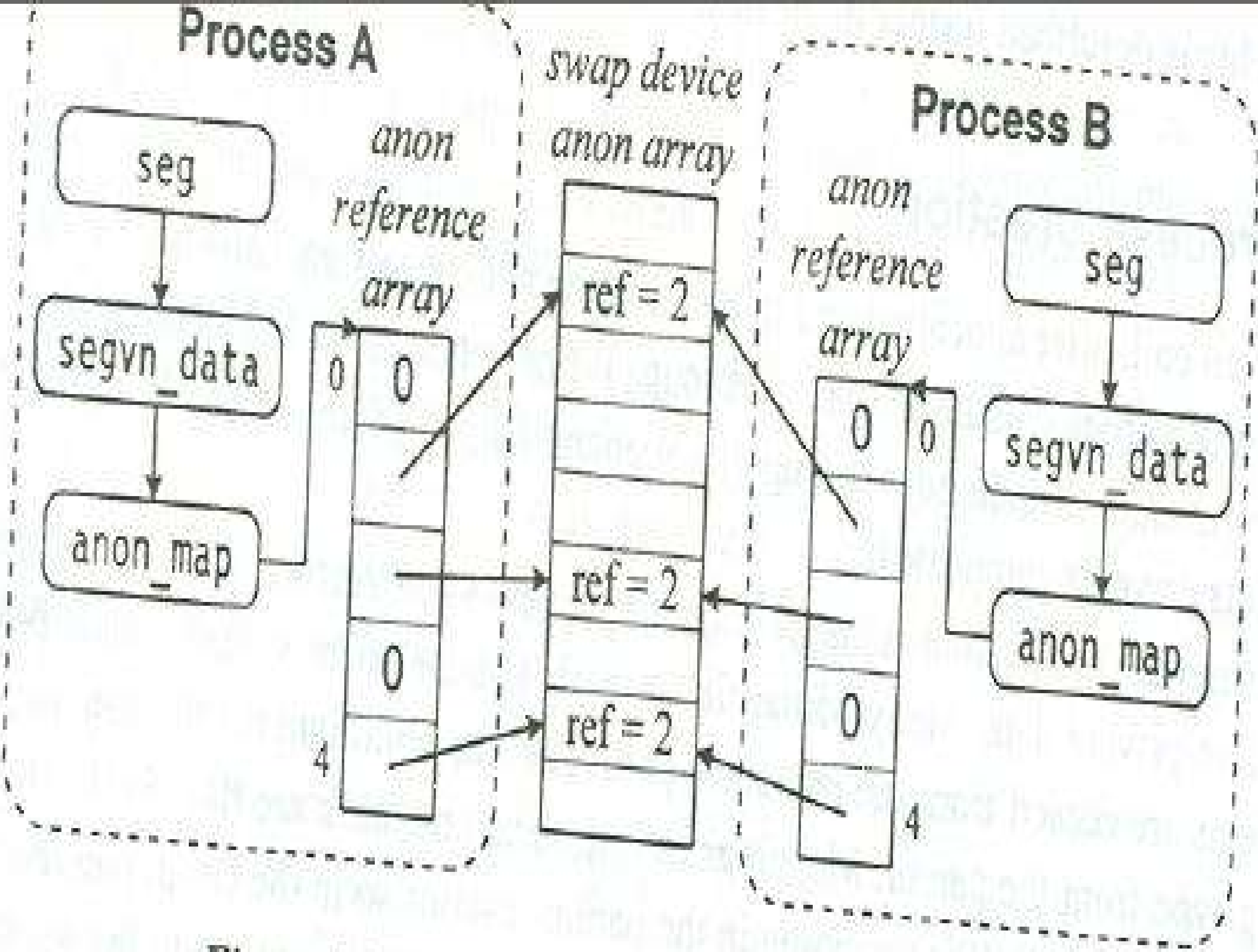


Figure 14-10. Sharing anonymous pages (part 1).

itially,

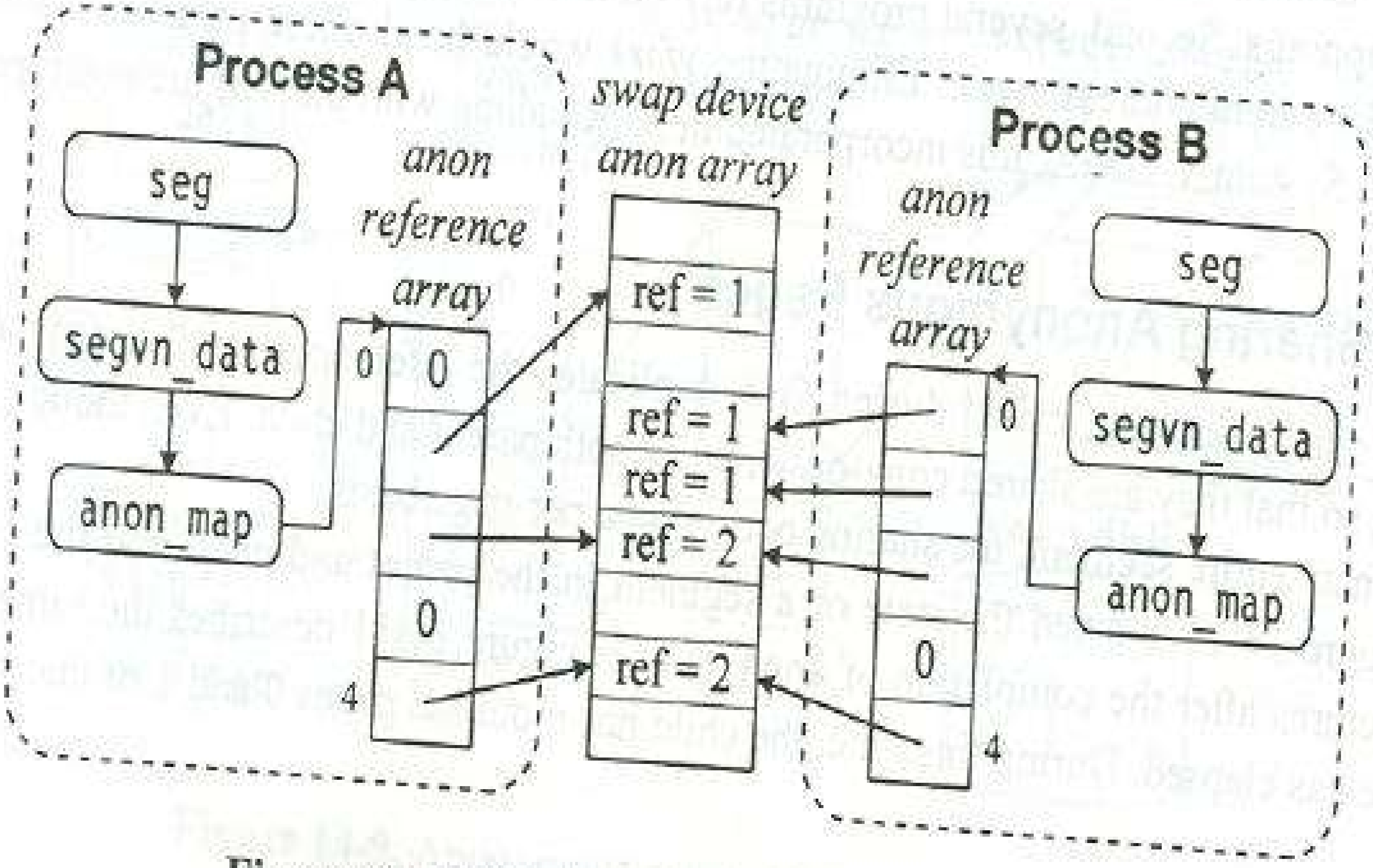


Figure 14-11. Sharing anonymous pages (part 2).

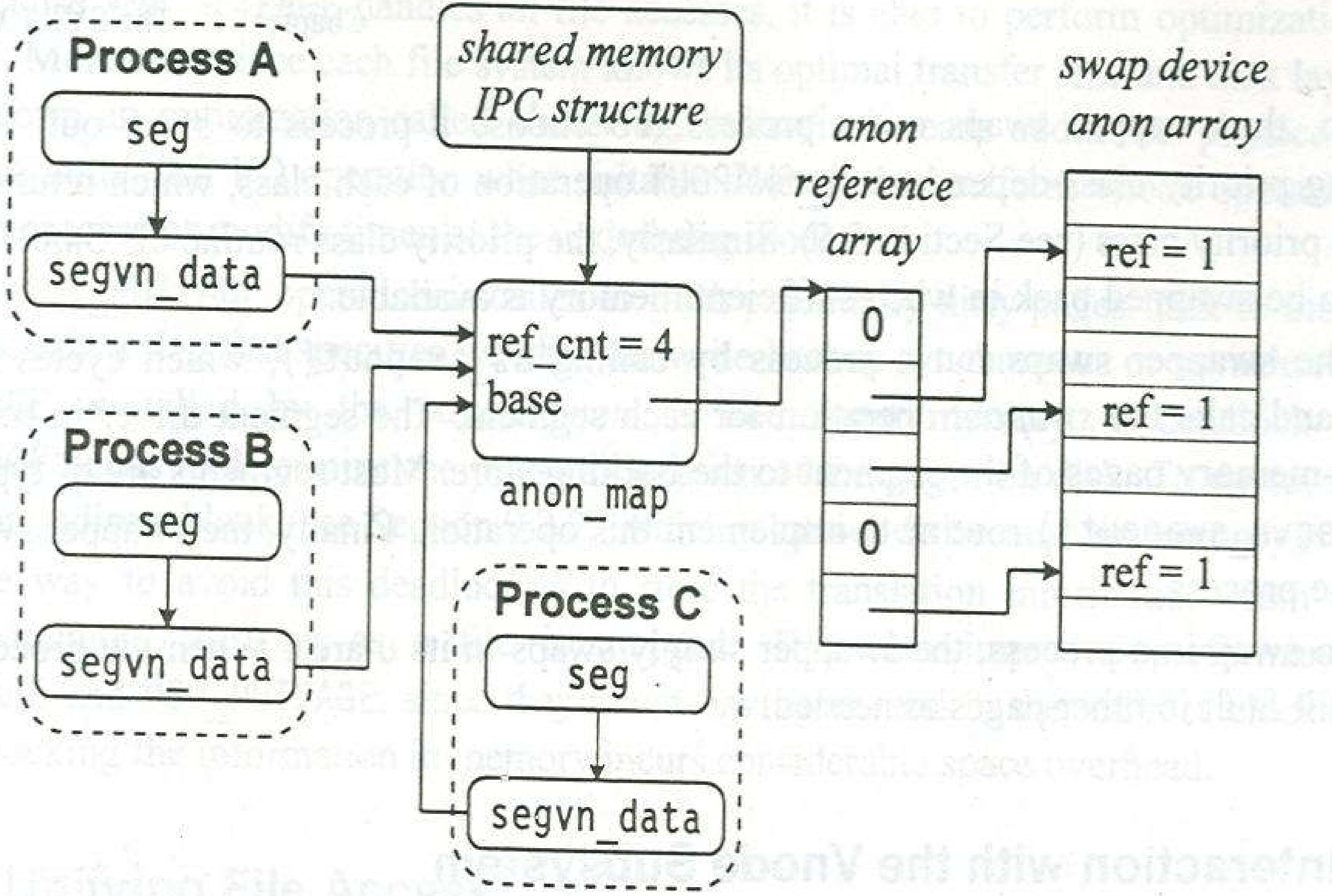


Figure 14-12. Implementing shared memory.

process creation: as_dup called: dupl parent's AS by as_alloc & SOP_DUP on segs

- parent's params (seg base, size, vp, offset, prots, seg_ops, mapping) copied
- text, data, stack: MAP_PRIVATE in both
- MAP_SHARED seg in parent: same in child
- data structs (anon_map/array) created lazily on mod (eg: for debugging text)
- fork: anon pages shared COW with child (anon_dup) with indep anon_map/array
    - call hat_chgprot to write-protect all anon pages
    - next call anon_dup for new anon_map & anon ref array; then clones all the refs in array by copying ptrs and incr ref counts
    - fault on first write!

page fault handling: trap routine for both prot and val faults

- as_fault called; uses hint or searches the sorted segments to locate seg
- SOP_FAULT called (seg_vn: segvn_fault); conv fault addr to logical pn in seg; check if real or spurious fault (send SIGSEGV if real)
    - spurious: to do COW or reference bit simulation
- pn used as index into anon_map & per-page prot array
    - if anon_map entry exists: anon_getpage (may find it in mem)
    - elsif seg mapped to file: VOP_GETPAGE (may find it in mem)
    - elsif mapped to anon obj: anon_zero to return zero-filled page
- _getpage handles special cases:
    - if page on free list, reclaim from list
    - if page inval for ref bit sim, reclaim by making page val
    - if page in transit, wait for I/O to complete
    - if page may already be in mem: search hashQs thru <vnode, offset>
- page now in memory; have to check for COW, etc.

How COW?

- If write access to private mapping &
- seg/per-page prots allow write &
- Page has no anon struct (private mapping to file) or its ref count >1 (COW on anon pages after fork)
- Call anon_private to privatise page
- Then call hat_memload to load new translation for the page into page tbls/TLBs

SVR4 VM

- Modular (seg drivers), portable (hat layer)
- Supports various forms of sharing: COW for indiv pages, MAP-SHARED mapping to anon obj for shared mem, shared access to files thru mmap, eff thru mmap, supports shared libs, supports execution of remote files (thru vnodes), integrated buffer cache+VM, debugging with breakpoints (text MAP_PRIVATE)
  - VM arch uses vnode ops for all file and disk accesses:
    - does not require special support for exec'ing NFS binaries. Swap devices can be on remote nodes=> diskless op
- But: more info/page, swap fragmentation & fs overhead, deadlocks possible with VOP_PUTPAGE (have to wire down indirect blocks to avoid it), more complex/slower; COW may be slower than informed prefetching

Page Replacement Algs

- FIFO: Belady's Anomaly
  - Reference string: 123412512345
  - No. of Faults as function of page frames:12(1 page frame), 12(2), 9(3), 10(4), 6(5), 6(6)
  - Not a stack alg! As fault(k)<=fault(k-1) violated
  - Uses the time when page enters mem as priority
- Optimal (unrealizable): victimize pages referred farthest in future
  - Uses the time when page is to be used
- FIFO second chance (= FIFO if all pages referenced)
  - Maintain circular Q and a ref bit for each page & a pointer
  - On page fault, from ptr, clear ref bits of referred pages & pick first page with ref bit 0

- Enhanced second chance or NRU: Ref & Mod (R &M) bits; clock interrupt clears R bit for all; victimization order: (~R, ~M), (~R,M) , (R, ~M), (R,M)
- LRU or NFU: Strict LRU also unrealizable
  - keep ctr for each page
  - on clock interrupt, ctr shifted right; ref bit copied to high bit, lsb dropped
  - Victimize page with least ctr value
- Page Buffering: pool of free frames, put victim into free list, if req page avlbl on free list, reuse; clean dirty pages during idle time