7.3 Operating System Issues

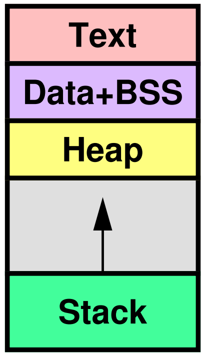
- *General Concerns*
- Representative Systems
- Copy on Write and Fork
- Backing Store Issues

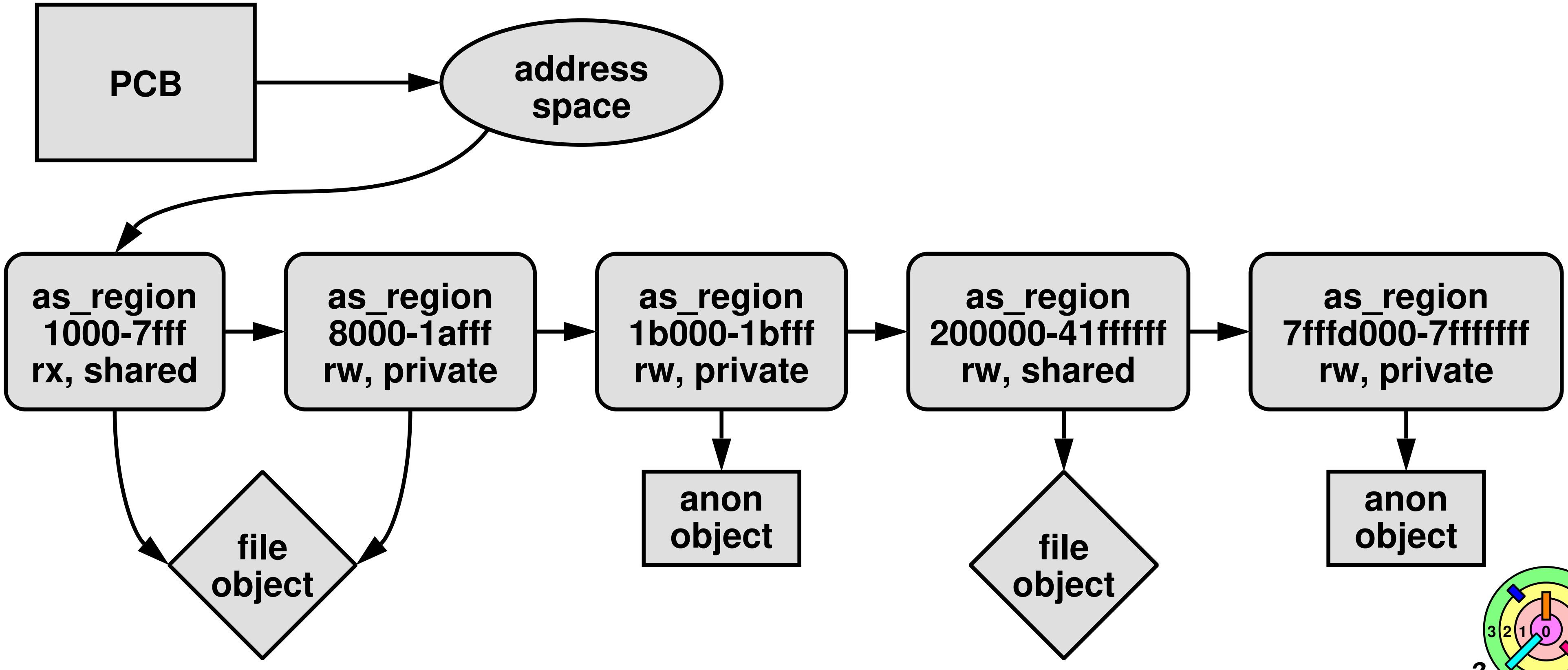
Virtual Memory: Traditional OS Issues

- Fetch policy
- Placement policy
- Replacement policy

Virtual Memory: Traditional OS Issues

- Fetch policy
- Placement policy
- Replacement policy





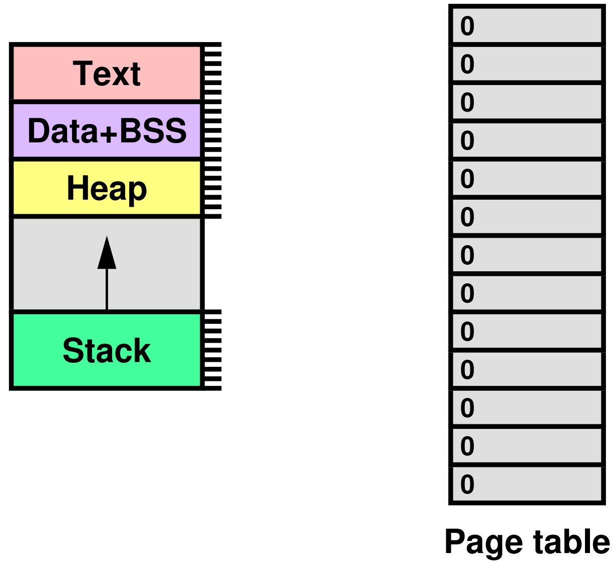
A Simple Paging Scheme

- ***Fetch*** **policy**
  - start process off with no pages in primary storage
  - bring in pages ***on demand*** (and only on demand)
    - this is known as ***demand paging***
      - defer processing until you absolutely have to do it
      - why? because you may not have to process ***at all***
      - ***demand paging*** is an instance of ***Lazy Evaluation***, a powerful idea used in computer science

A Simple Paging Scheme

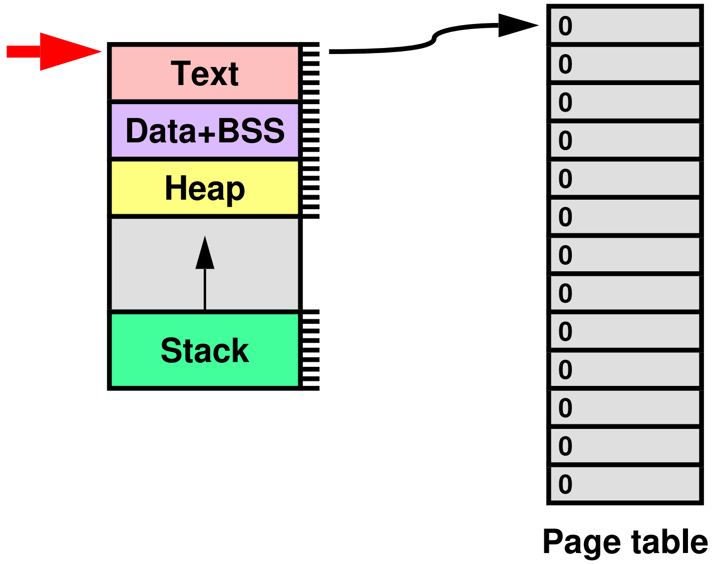
- ***Placement*** **policy**
  - **unlike disk pages, it doesn't matter here - put the incoming page (from disk) in the first available physical page**
    - **page *frames* are used to keep track of physical pages**
- ***Replacement*** **policy**
  - **required if there is not enough *resource* to go around**
  - **e.g., replace the page that has been in primary storage the longest (FIFO policy, which can be bad)**

Demand Paging



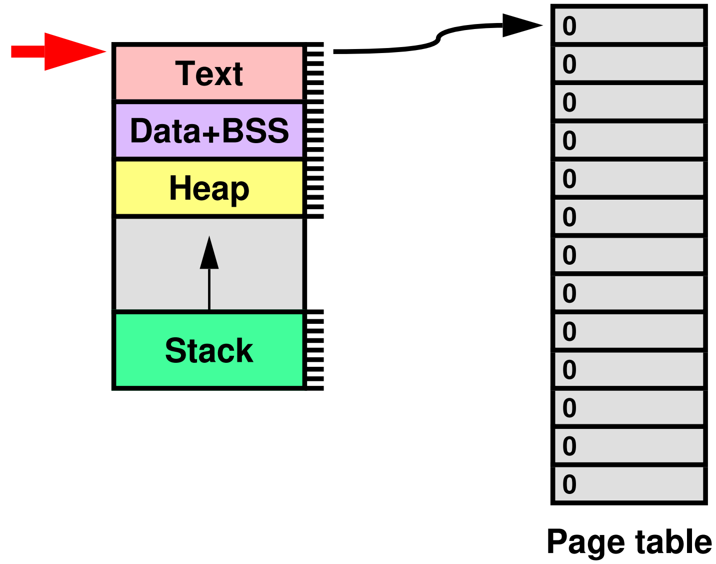
- In `exec()`, address space is created and page table is cleared with all entries having V=0

Demand Paging



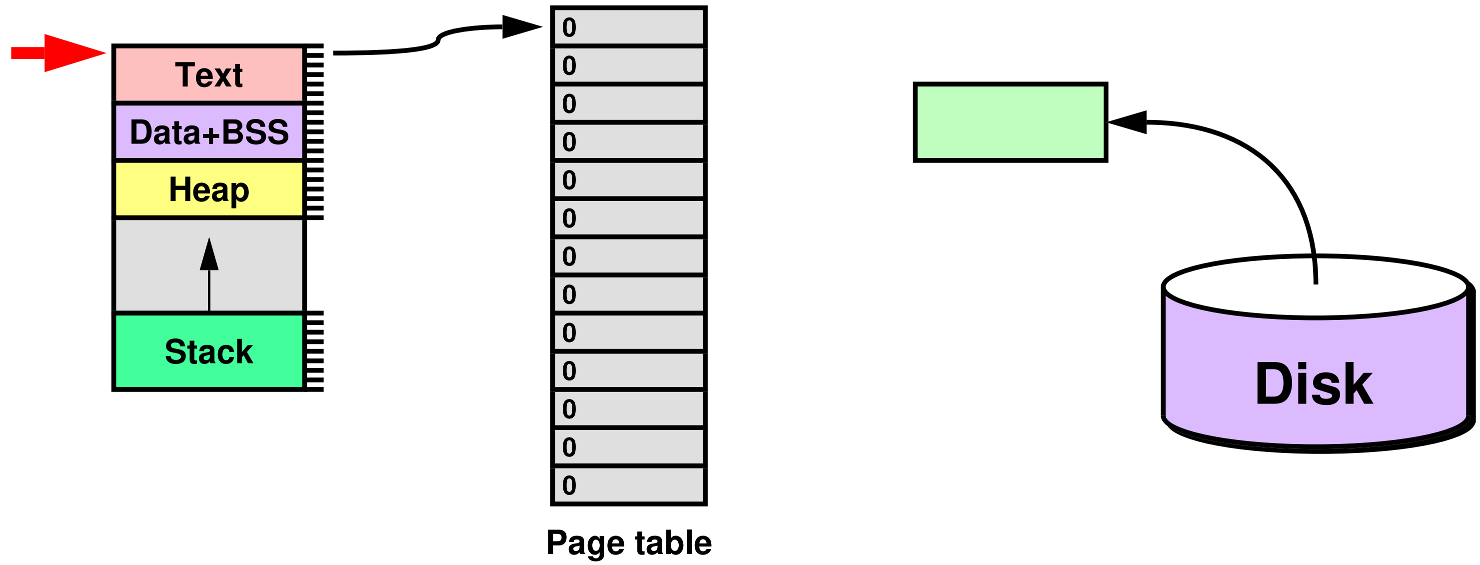
- In `exec()`, address space is created and page table is cleared with all entries having V=0
  - as the *first instruction executes*

Demand Paging



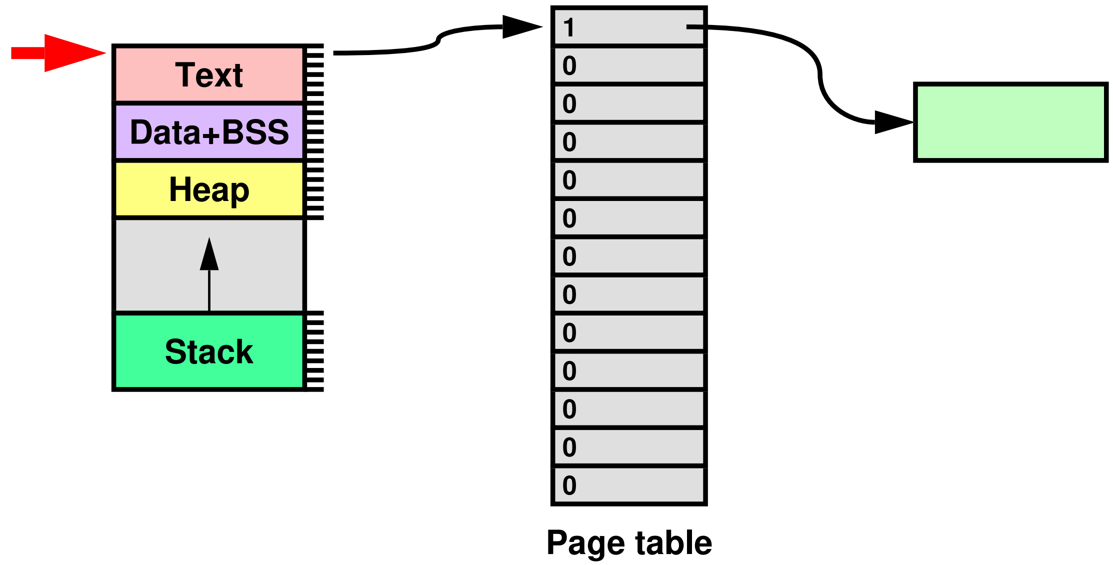
- In `exec()`, address space is created and page table is cleared with all entries having V=0
  - as the *first instruction executes*
    - since V=0, the hardware traps into the kernel

Demand Paging



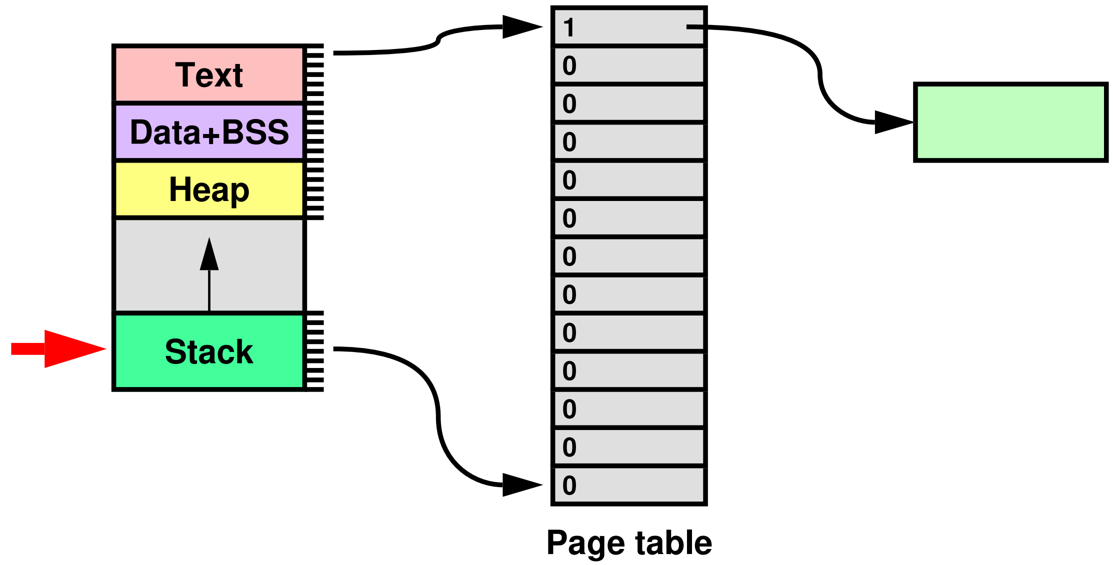
- In `exec()`, address space is created and page table is cleared with all entries having V=0
  - as the *first instruction executes*
    - since V=0, the hardware traps into the kernel
    - the kernel *allocates a physical page* and copy the first 4KB of code into this page (allocate from where?)
      - point the corresponding page table entry to this page
      - update all necessary data structures

Demand Paging



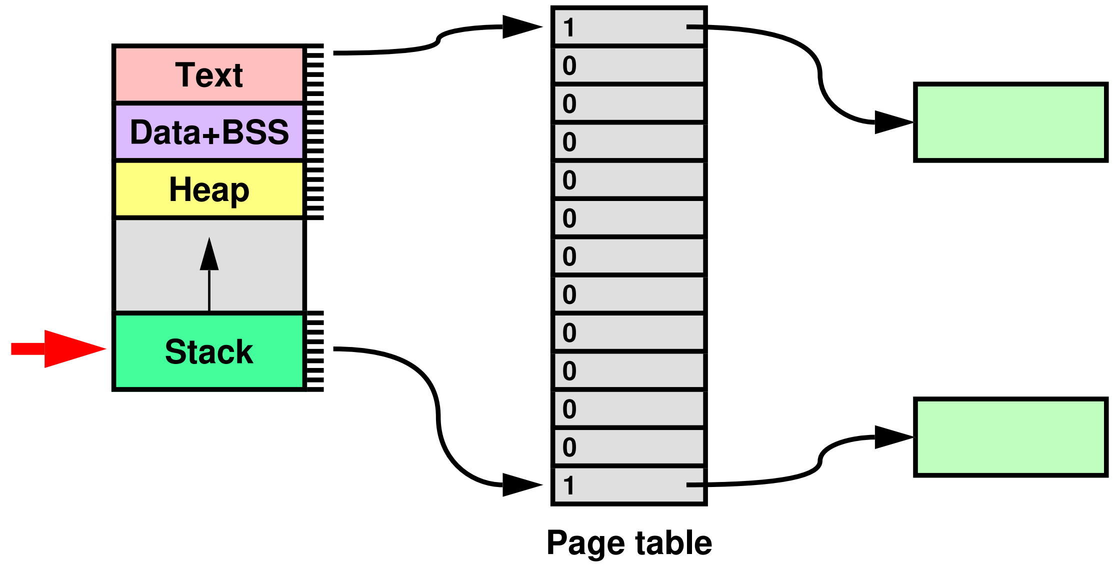
- In `exec()`, address space is created and page table is cleared with all entries having V=0
  - as the *first instruction executes*
    - since V=0, the hardware traps into the kernel
    - the kernel *allocates a physical page* and copy the first 4KB of code into this page (allocate from where?)
      - point the corresponding page table entry to this page
      - update all necessary data structures
    - set V=1 and return from the trap

Demand Paging



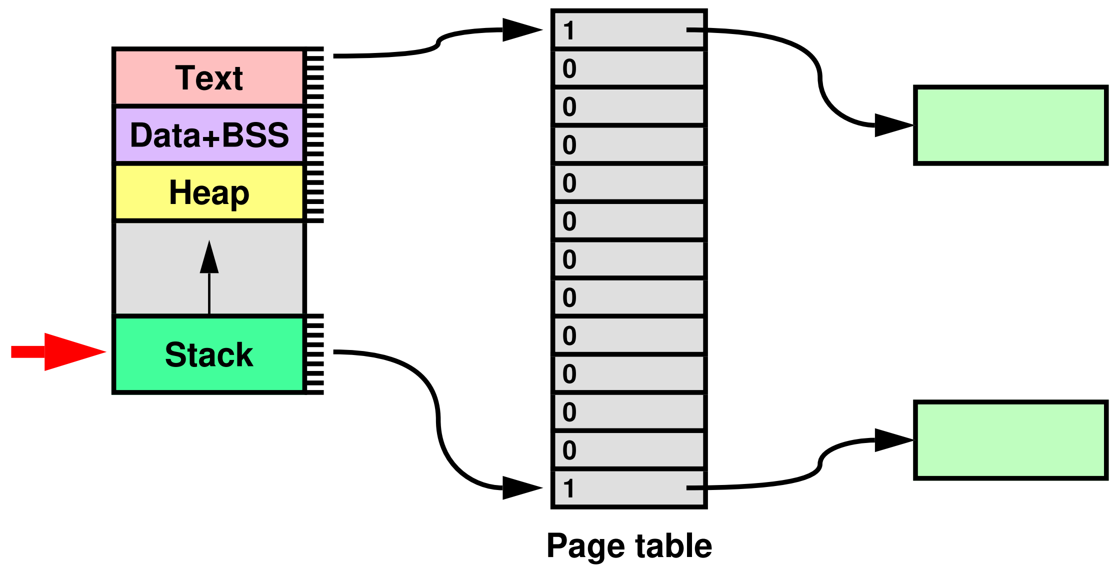
- In `exec()`, address space is created and page table is cleared with all entries having V=0
  - as the program access the ***stack***

Demand Paging



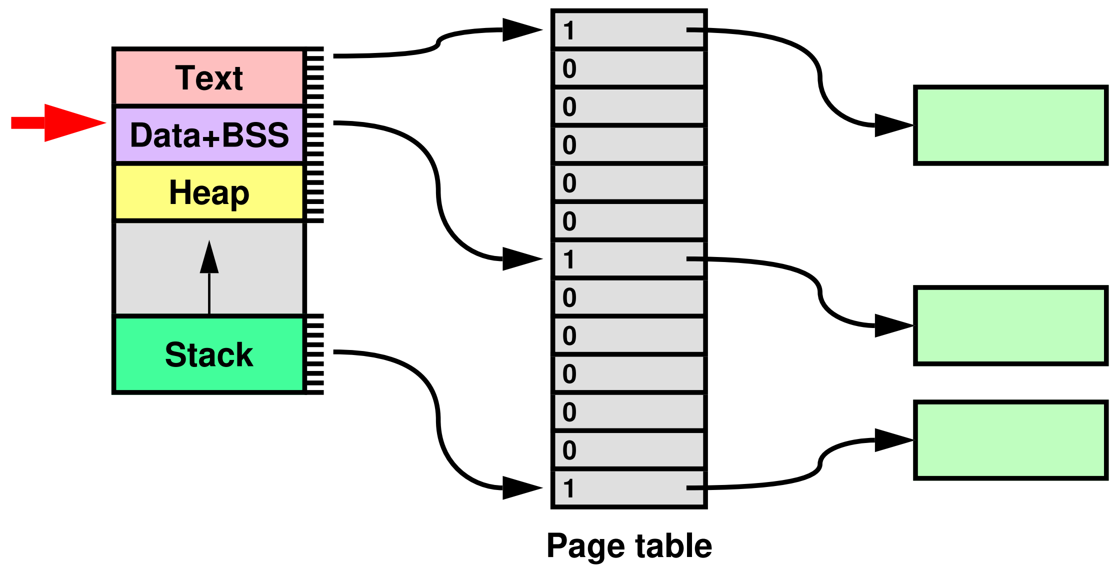
- In `exec()`, address space is created and page table is cleared with all entries having V=0
  - as the program access the ***stack***
    - similar things happen

Demand Paging



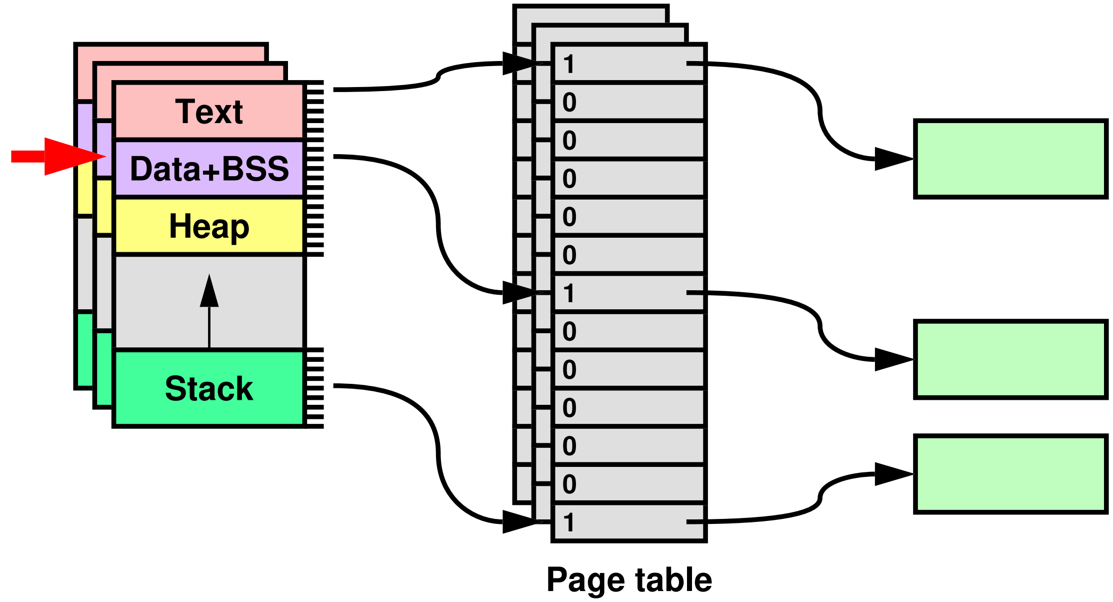
- In `exec()`, address space is created and page table is cleared with all entries having V=0
  - as the program access the *stack*
    - similar things happen
    - although stack is a little different since it needs a *backing store* and need to set up for *copy-on-write*

Demand Paging



- In `exec()`, address space is created and page table is cleared with all entries having V=0
  - as the program access the *stack*
    - similar things happen
    - although stack is a little different since it needs a *backing store* and need to set up for *copy-on-write*
  - accessing the *data segment* is similar to stack (but different)
    - original (read-only) backing store is the executable file
    - after copy-on-write, backing store is the swap space

Demand Paging



- Remember, there are multiple processes and multiple page tables that the OS is servicing
  - complicated by the fact that page frames can be shared

- In kernel 3, you need to make sure that every time when you return back into user space, all kernel data structures are in a consistent state

Page Fault

- ***Page Fault*** (accessing a page with ***V=0***)
  1) Trap occurs (due to a page fault)
  2) Find free physical page
  3) Write page out if no free physical page
  4) Fetch page
  5) Return from trap

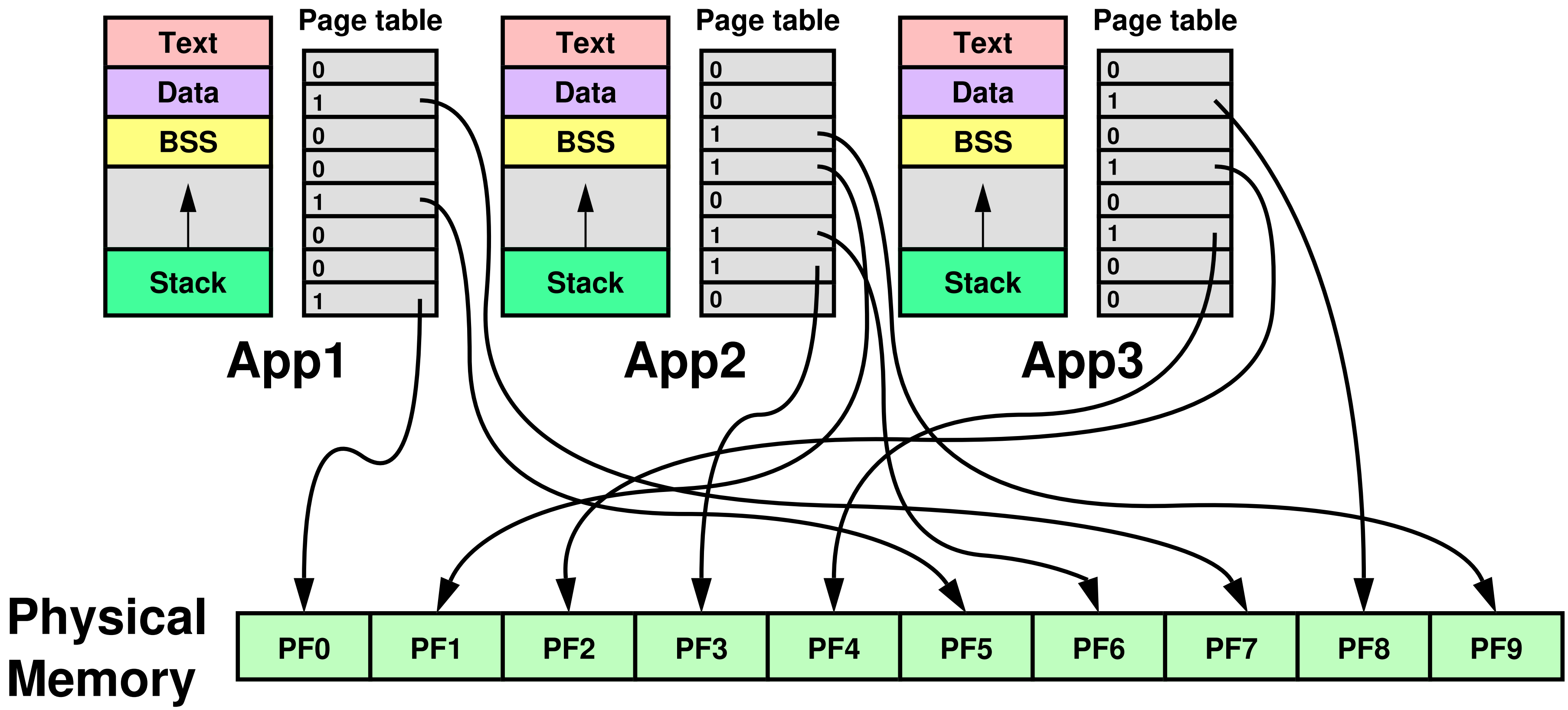
- **Issues**
  - in step (2), where and how do we find such a ***free physical page***?

Page Fault

- ***Page Fault*** (accessing a page with ***V=0***)
  1) Trap occurs (due to a page fault)
  2) Find free physical page
  3) Write page out if no free physical page
  4) Fetch page
  5) Return from trap

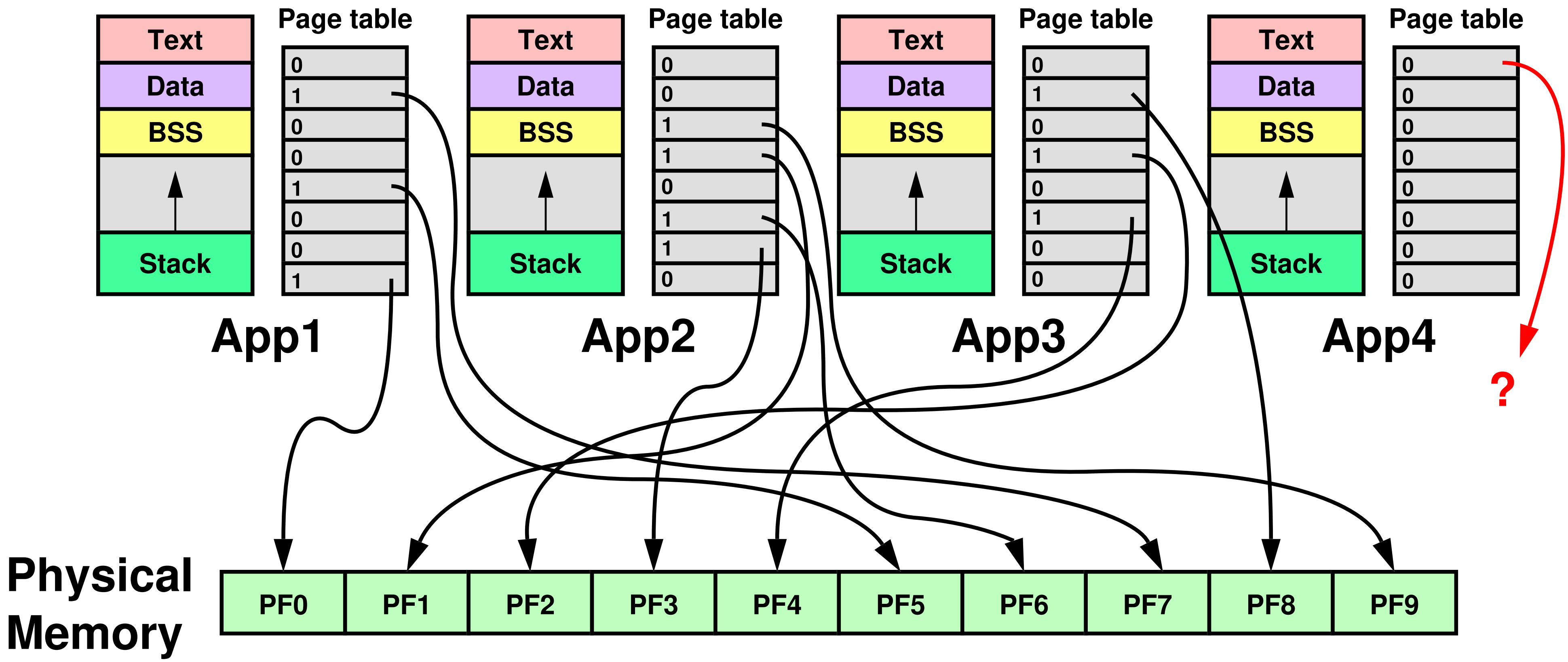
- **Issues**
  - in step (2), where and how do we find such a ***free physical page***?
    - the Buddy System is used
      - return NULL if no free physical page is available
  - in step (3), where and how do we find an ***in-use physical page*** to write out to disk?

Example



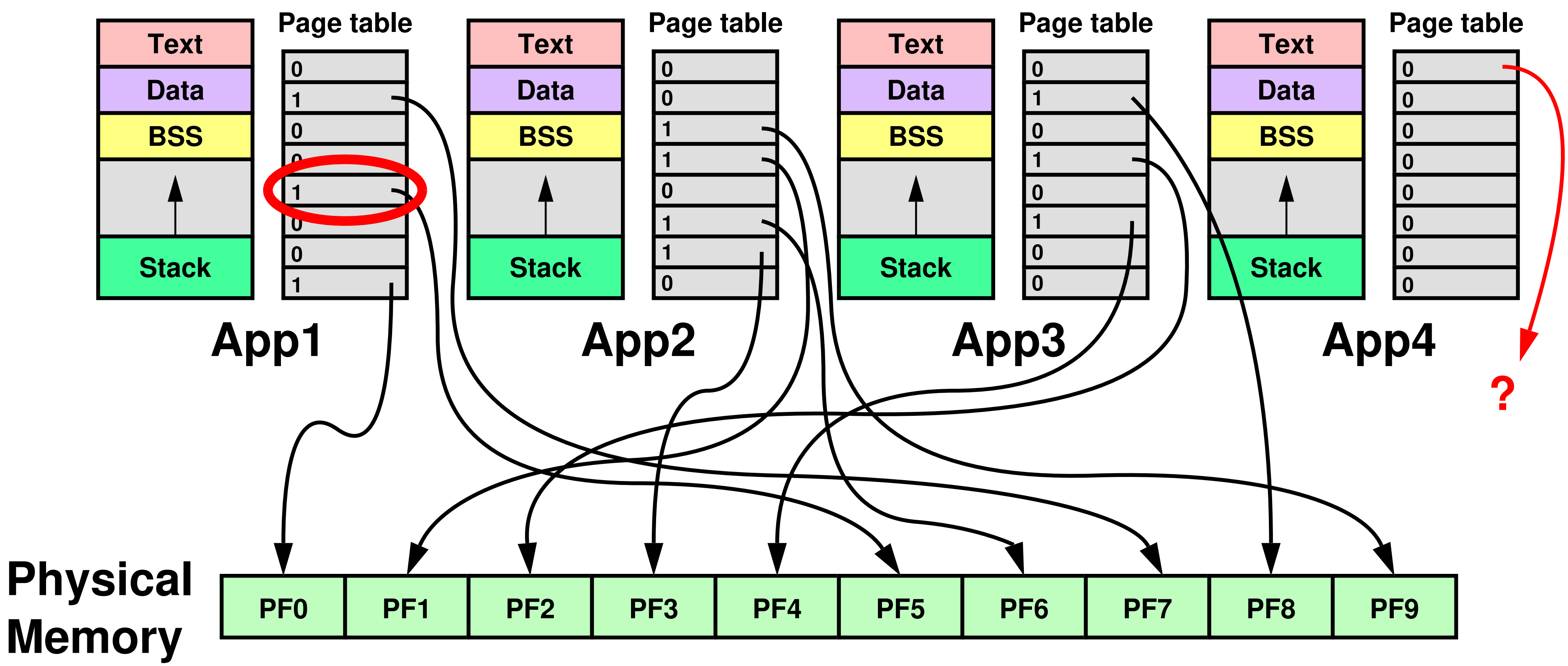
Text
Data
BSS
Stack
Page table
0
1
0
0
1
0
0
1
App1
Text
Data
BSS
Stack
Page table
0
0
1
1
0
1
1
0
App2
Text
Data
BSS
Stack
Page table
0
1
0
1
0
1
0
0
App3
Physical Memory
PF0
PF1
PF2
PF3
PF4
PF5
PF6
PF7
PF8
PF9

Example



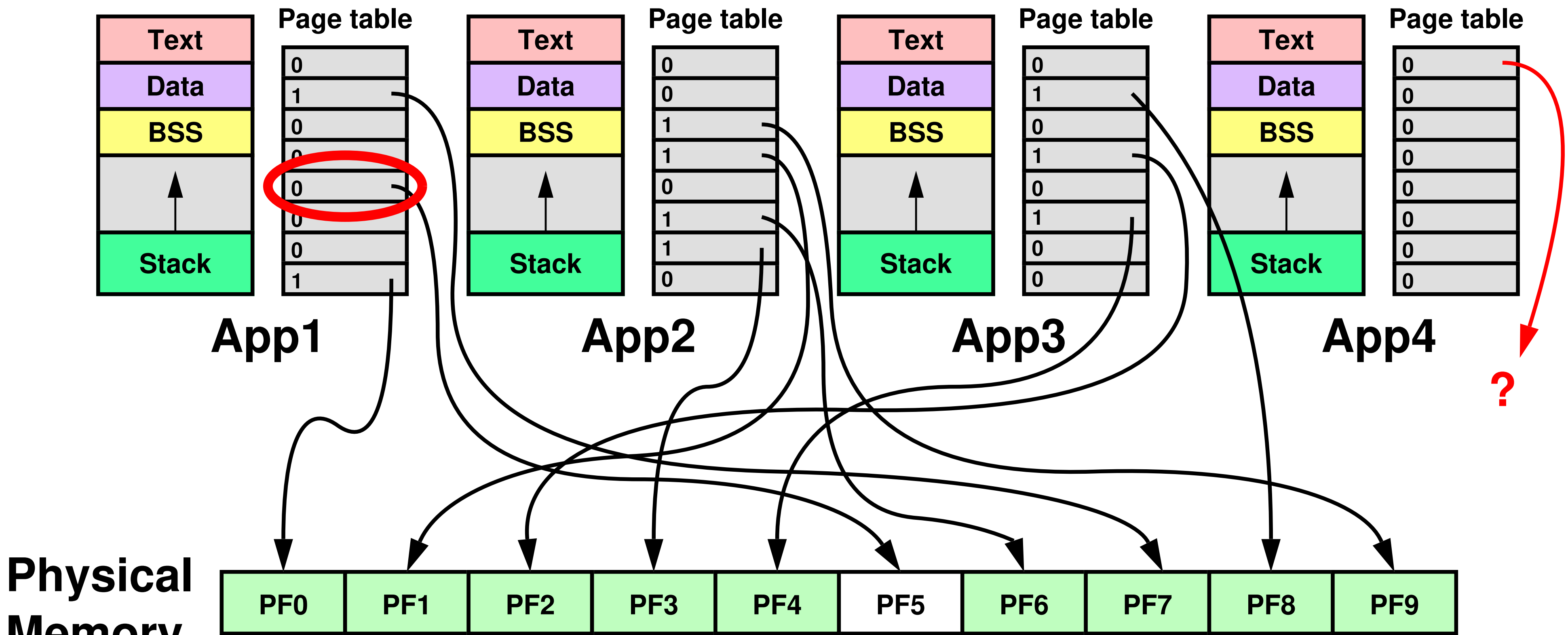
- Need a physical page
  - all physical pages are in use

Example

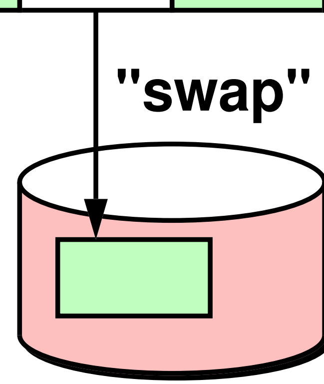


- **Need a physical page**
  - all physical pages are in use
  - pick any physical page
    - well, according to the *page replacement policy*

Example

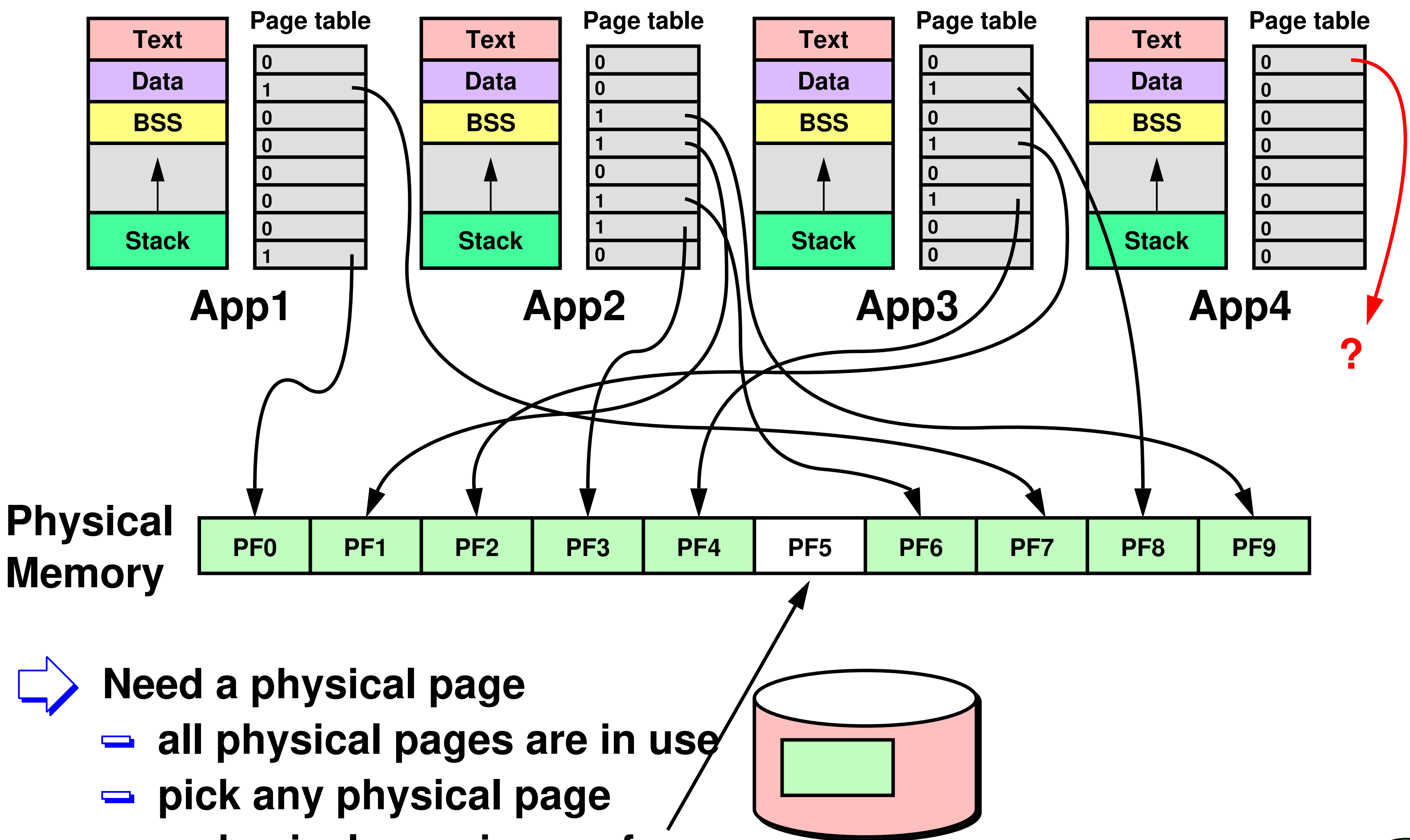


"swap" this physical page out into its "backing store" *(write to disk if the page frame is "dirty")*



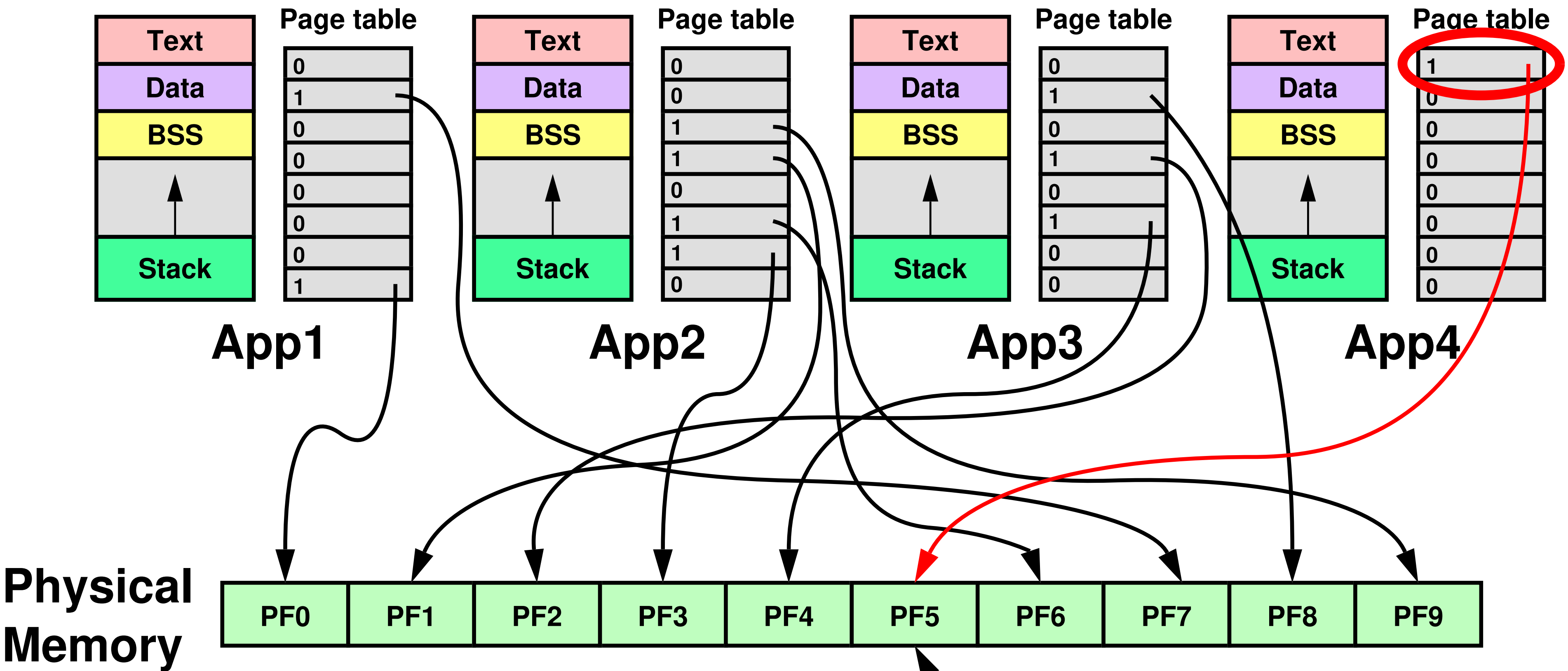
- Need a physical page
  - all physical pages are in use
  - pick any physical page
    - kernel keeps track of where the physical page is copied to

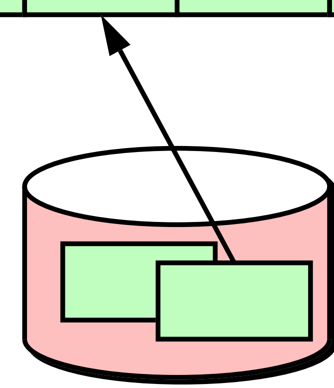
Example



- Need a physical page
  - all physical pages are in use
  - pick any physical page
  - a physical page is now free

Example





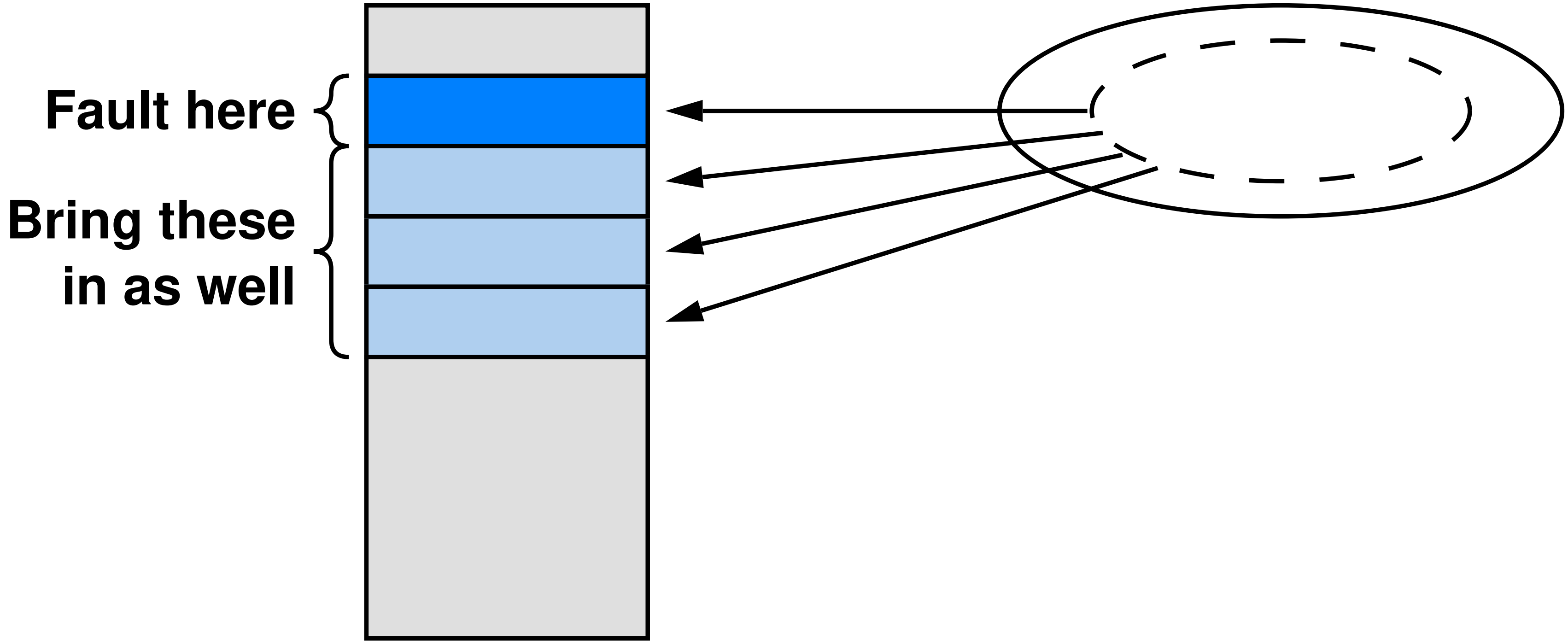
- Need a physical page
  - all physical pages are in use
  - pick any physical page
  - a physical page is now free
  - fetch page from disk and fix up page table

Performance

- *Page Fault* (accessing a page with *V=0*)
  1) Trap occurs (due to a page fault)
  2) Find free physical page
  3) Write page out if no free physical page
  4) Fetch page
  5) Return from trap

- A page fault can result in disk operations and slow down the application
  - do not want to wait for the disk!
  - need to reduce this latency
    - *prefetching*
    - *pageout daemon*

Improving the Fetch Policy

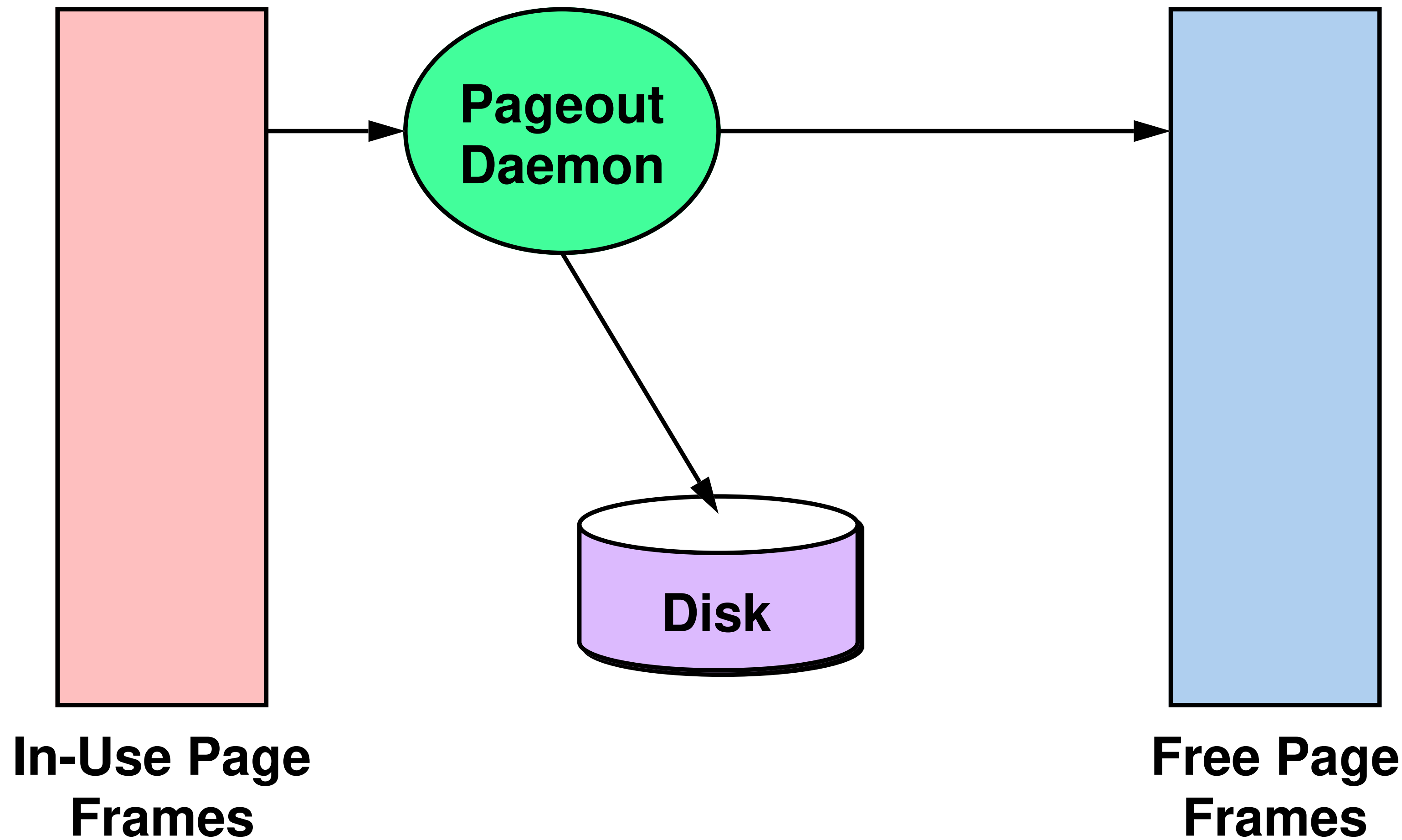


- This is *prefetching*
  - accesses to pages is often sequential
  - gamble that this is worthwhile (since it takes up more memory)
- This improves step (4) on previous page
  - but it uses up physical memory faster and makes (3) more likely to occur

Improving the Replacement Policy

- ***When*** **is replacement done?**
  - **doing it "on demand" causes excessive delays**
    - **so, "on-demand" (or Lazy Evaluation) is *not always* a good policy**
  - **should be performed as a separate, *concurrent* activity**
    - **use a thread *(i.e., a pageout deamon)* to *continuously* and *aggressively* look for free pages**

The "Pageout Daemon"

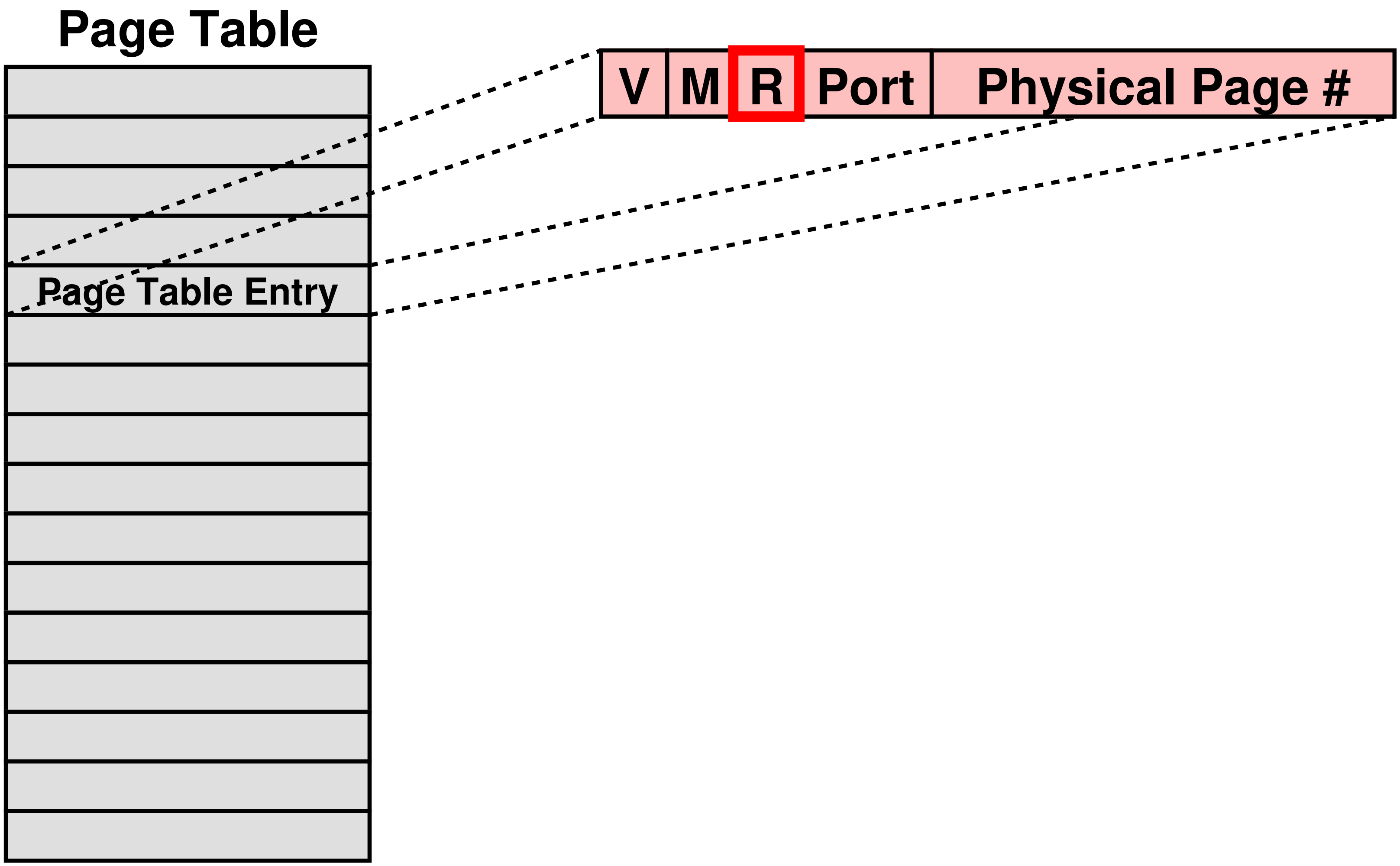


- ***Page frames*** are used to keep track of physical pages
- Can use ***multiple*** pageout daemons

Choosing the Page to Remove - Replacement Policy

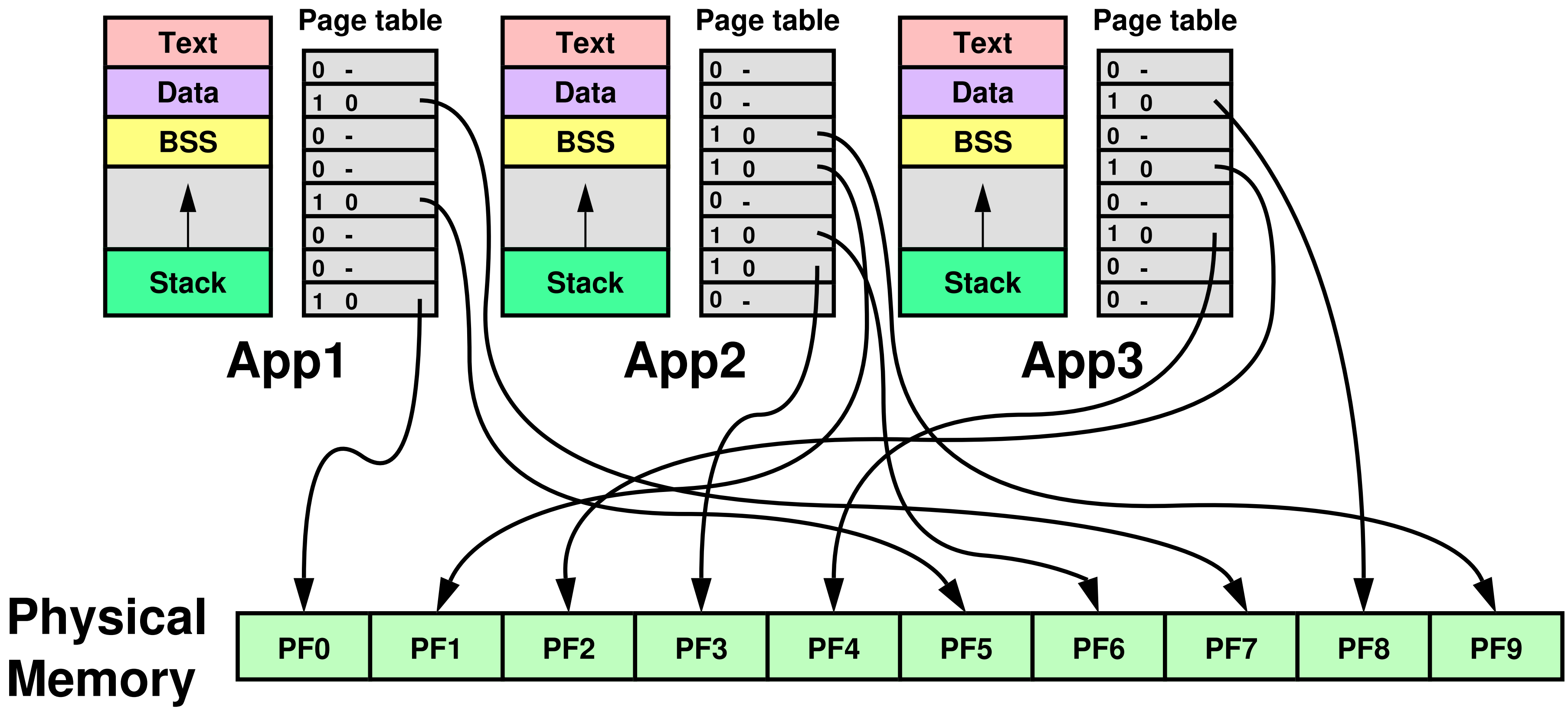
- *Which* pages are replaced?
  - FIFO policy is not good
  - want to replace those pages least likely to be referenced soon
- If your DVD rack is full and you just bought a new DVD
  - which DVD would you remove from the rack to make room for the new DVD?
- Idealized policies:
  - FIFO (First-In-First-Out)
  - LFU (Least-Frequently-Used)
  - LRU (Least-Recently-Used)

Implementing LRU



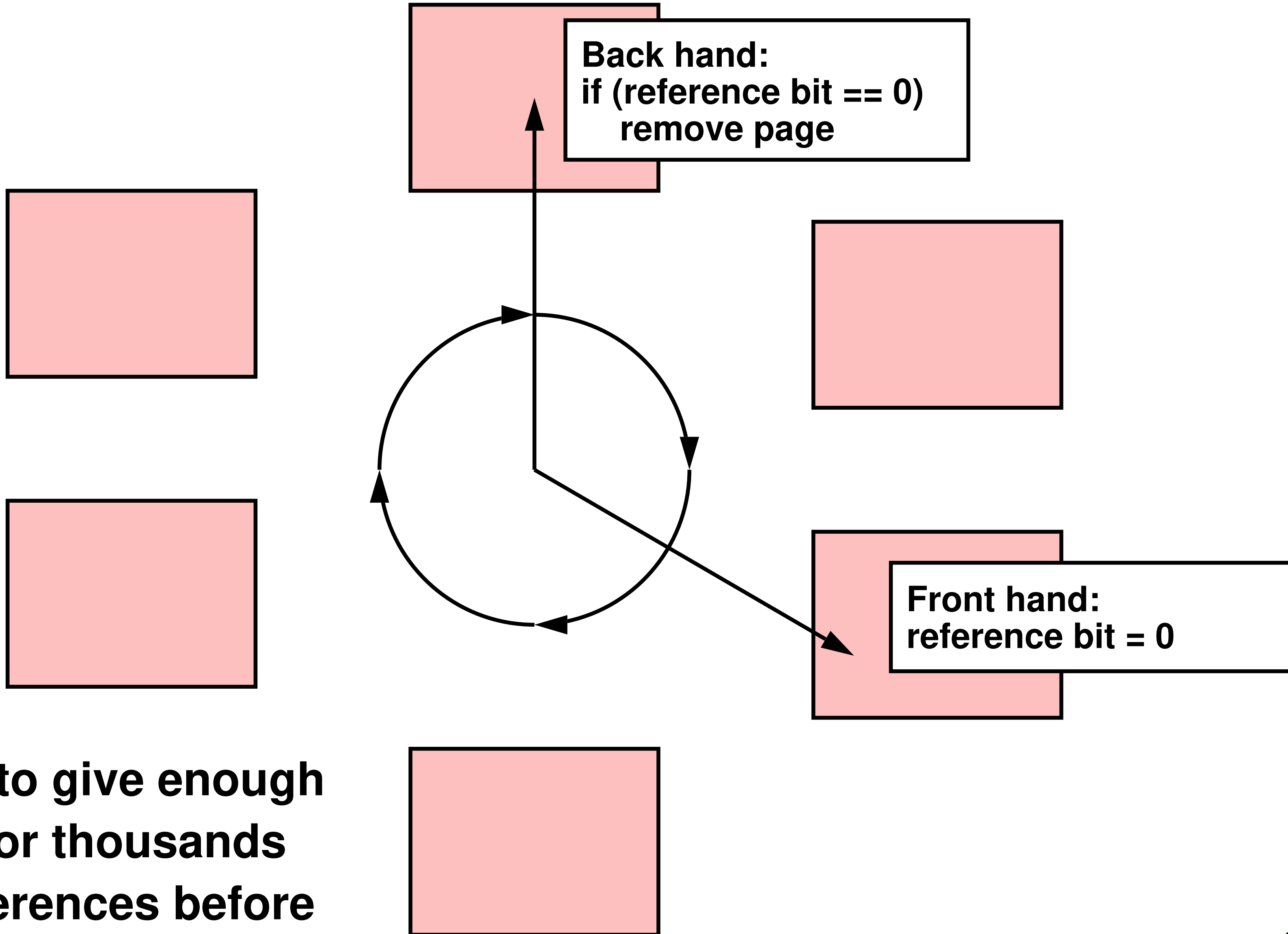
- To approximate LRU (a very coarse approximation), the *reference* bit in the page table entry is used

Using The Reference Bits



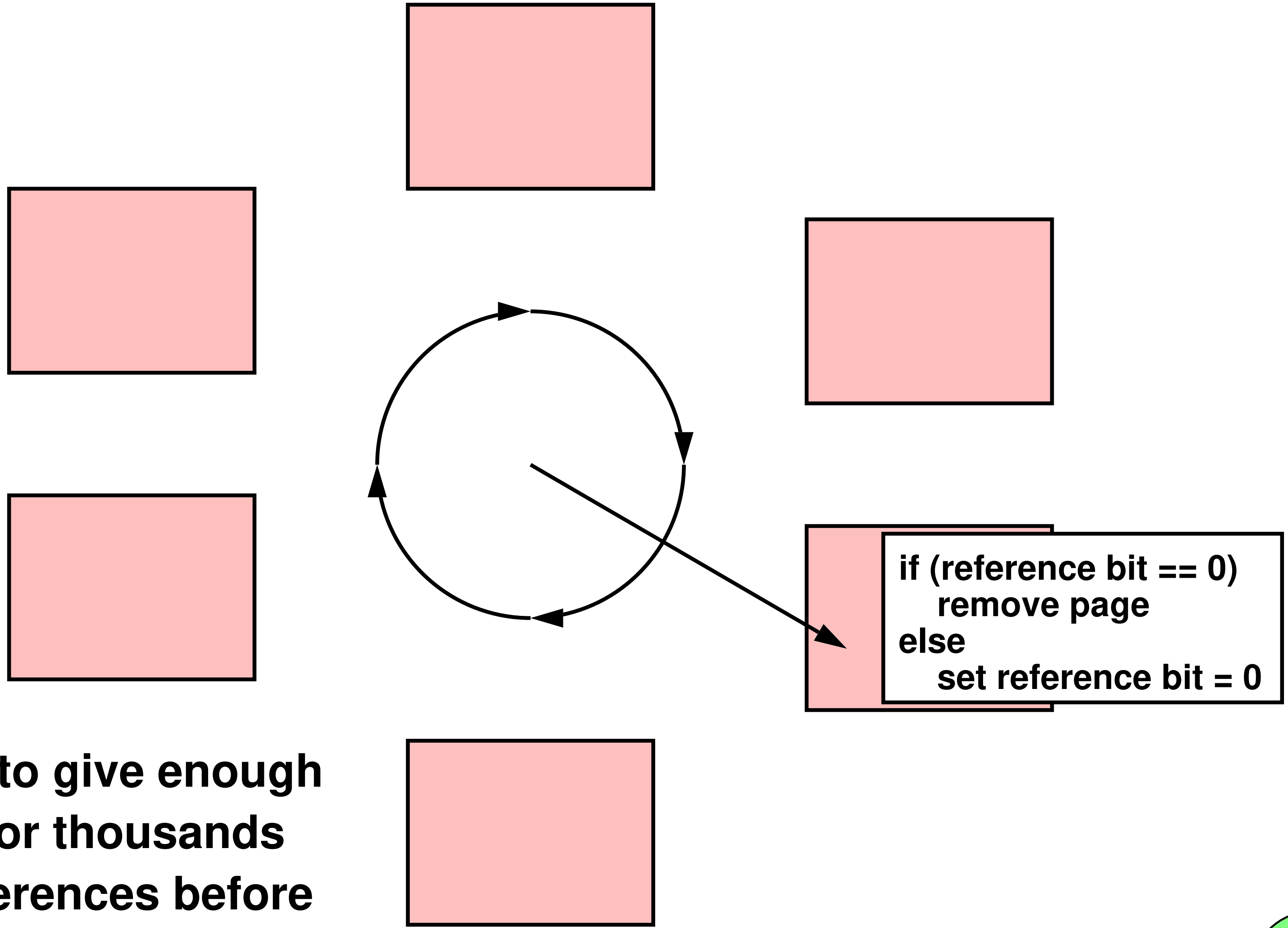
- Why would some pages referenced more often than other?
  - code?
  - stack?
  - depends on the application

Clock Algorithm - Two-handed



- Need to give enough time for thousands of references before checking

Clock Algorithm - One-handed

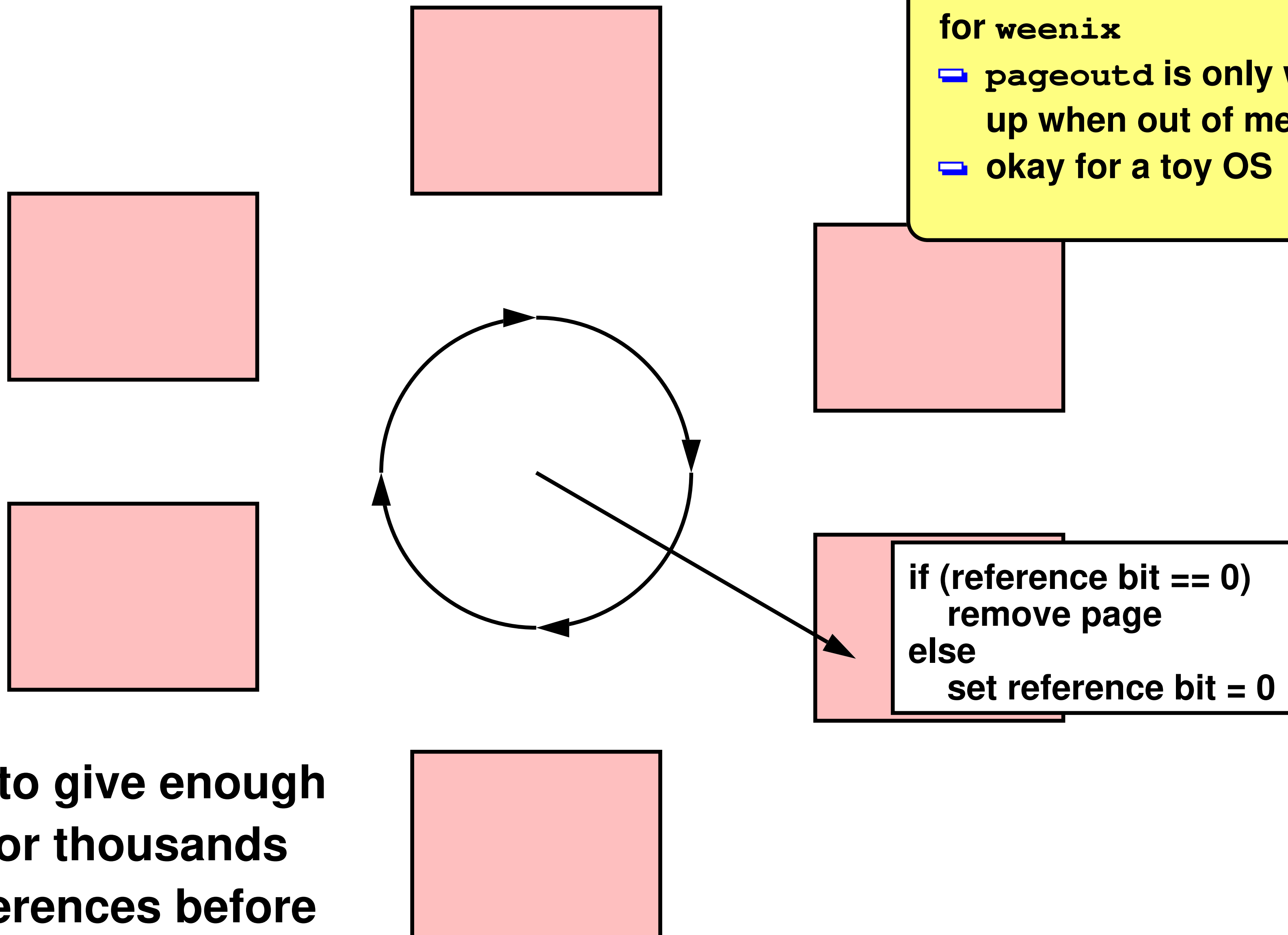


- Need to give enough time for thousands of references before checking

Clock Algorithm - One-handed

for **`weenix`**

- **`pageoutd`** is only woken up when out of memory
- okay for a toy OS



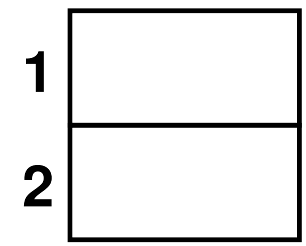
- **Need to give enough time for thousands of references before checking**

Global vs. Local Allocation

- What if a process uses up all the page frames?
- Global allocation
  - all processes *compete* for page frames from a single pool
  - problem:
    - memory-hungry processes will get all the memory
    - possibility of *thrashing*
- Local allocation
  - each process has its own *private* pool of page frames
  - Windows does this
    - processes do not have to compete for the same pool of page frames
    - goal is to *minimize the possibility of thrashing*

Thrashing

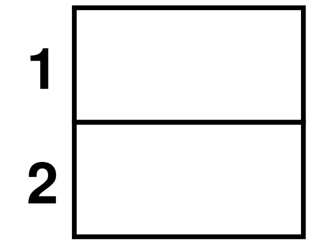
- Consider a system that has exactly two page frames:
  - process A has a page in frame 1
  - process B has a page in frame 2



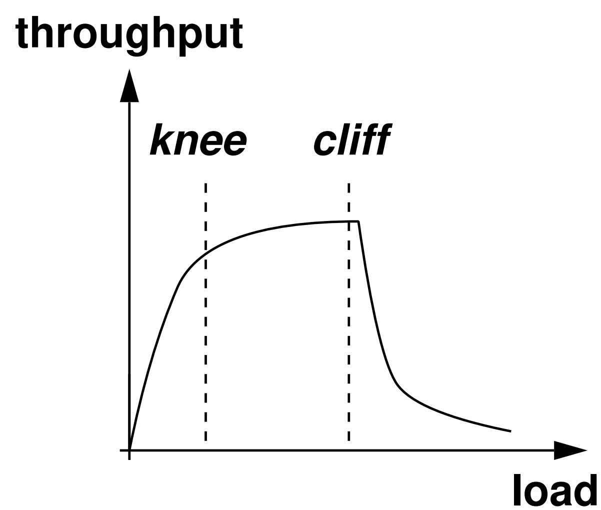
- Process A references another page, causing a page fault
  - the page in frame 2 is removed from B and given to A
- Process B faults immediately; the page in frame 1 is given to B
- Process A resumes execution and faults again; the page in frame 1 is given back to A
- ...
  - neither processes makes progress

Thrashing

- Consider a system that has exactly two page frames:
  - process A has a page in frame 1
  - process B has a page in frame 2



- Process A references another page, causing a page fault
  - the page in frame 2 is removed from B and given to A
- Process B faults immediately; the page in frame 1 is given to B
- Process A resumes execution and faults again; the page in frame 1 is given back to A
- ...
  - neither processes makes progress



- Although this is a contrived example, it highlights the basic problem
  - need 3 physical page frames, but only 2 are available

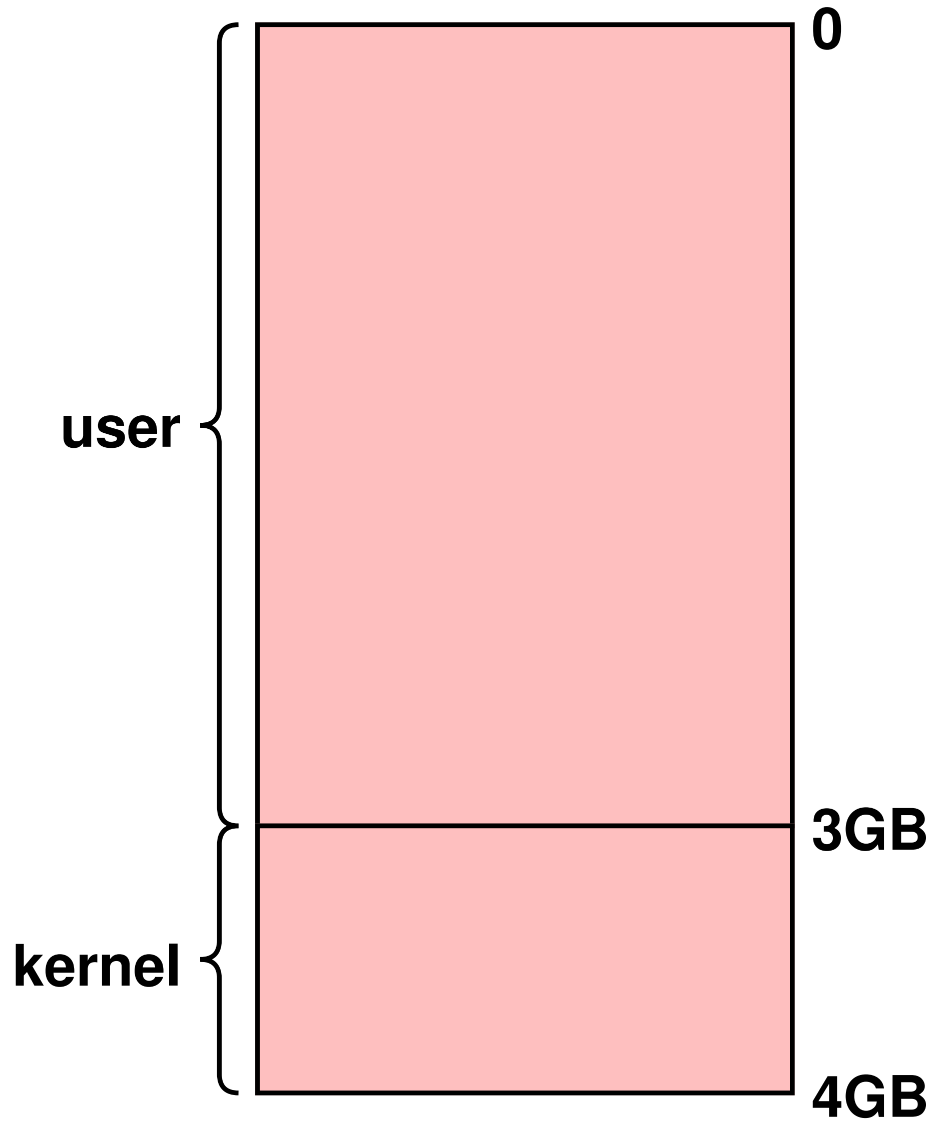
The Working-Set Principle

- To deal with thrashing, the idea of *Working-Set* can be used
  - although it may be difficult to implement exactly
- The set of pages being used by a program (the working set) is relatively small and changes slowly with time
  - *WS(P,T)* is the set of *pages* used by process *P* over time period *T*
- Over time period *T*, *P* should be given *|WS(P,T)|* page frames
  - if space isn't available, then *P* should not run and should be *swapped out*
- If the sum of the working-set of all processes is less than the total amount of available physical memory
  - then thrashing cannot occur
  - using *Local Allocation* is a way to *reduce the chance* of *thrashing*

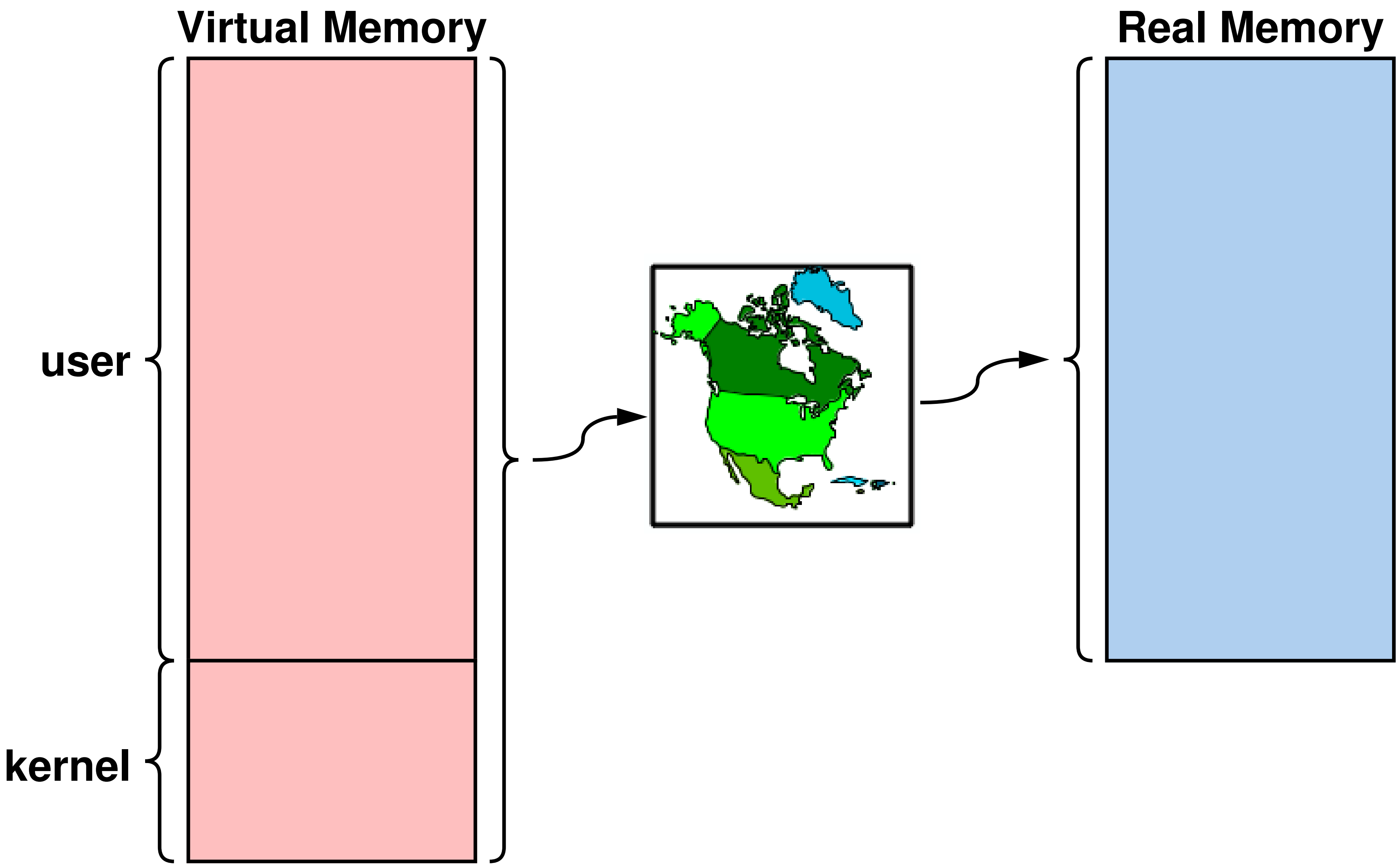
7.3 Operating System Issues

- General Concerns
- *Representative Systems*
- Copy on Write and Fork
- Backing Store Issues

Linux Intel x86 VM Layout



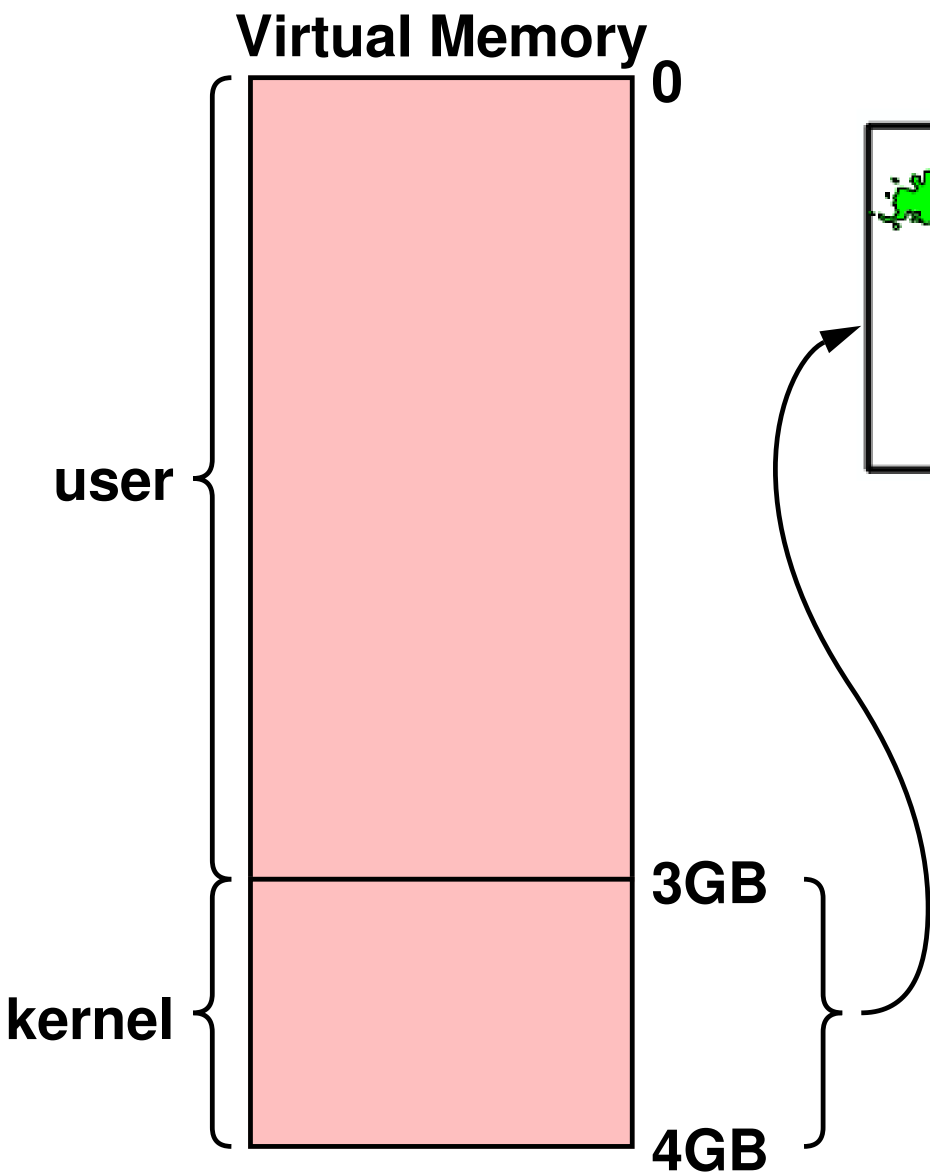
Real Memory

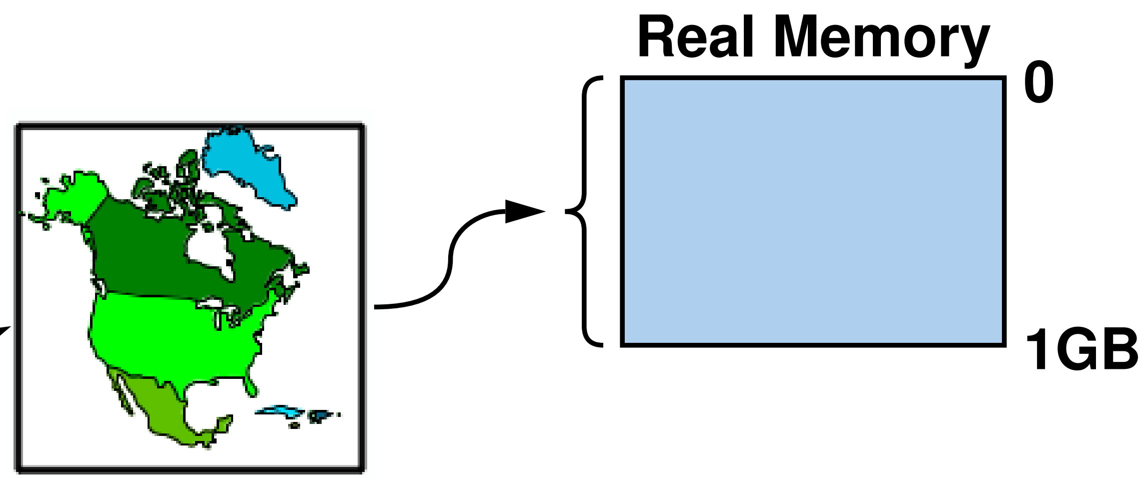


Memory Allocation

- User
  - virtual allocation
    - fork
    - pthread_create
    - exec
    - brk
    - mmap
  - real allocation
    - (not done)
- OS kernel
  - virtual allocation
    - fork, etc.
    - some kernel data structures
    - pretty much any time when you allocate from a slab allocator
  - real allocation
    - page faults
    - some kernel data structures
      - e.g., page tables
    - pretty much any time when you allocate from the buddy system

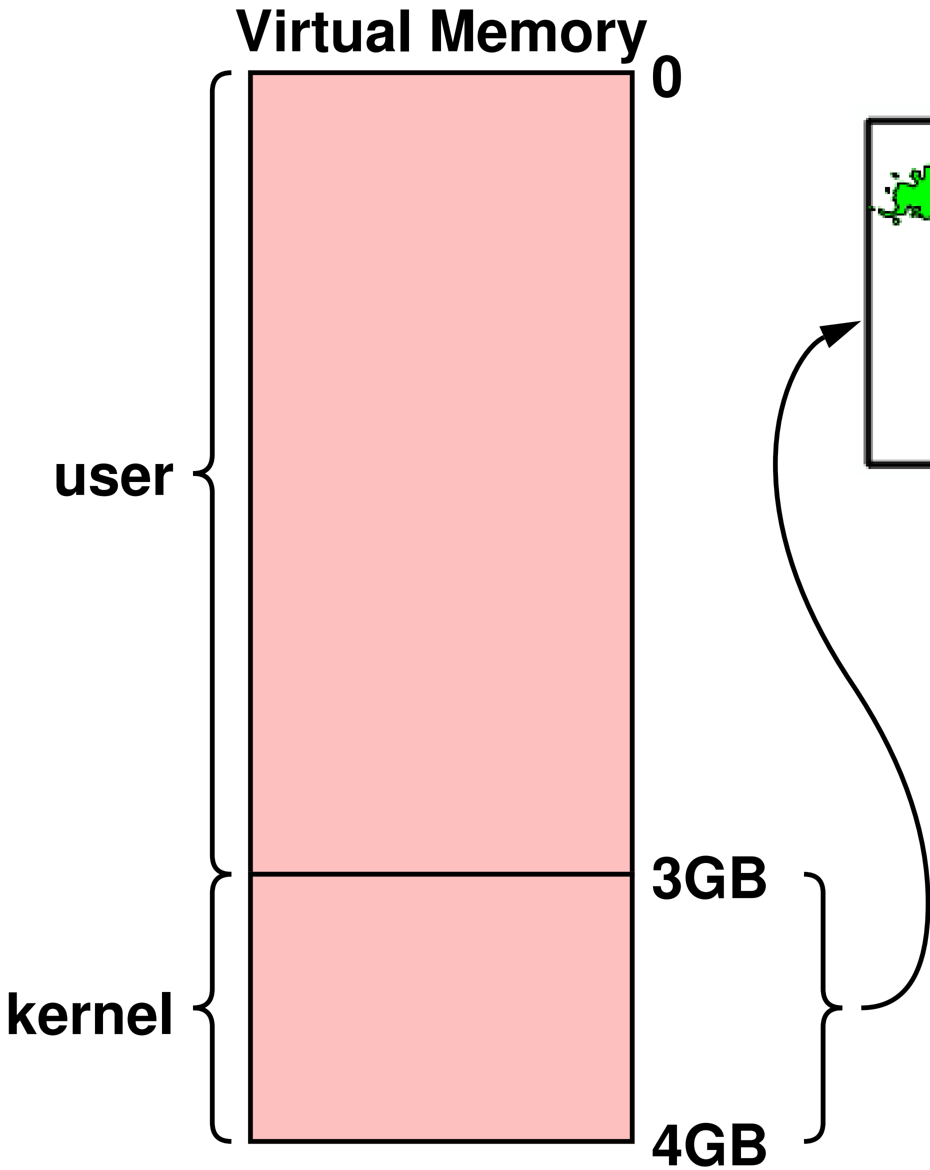
Example: 1GB Of Real Memory

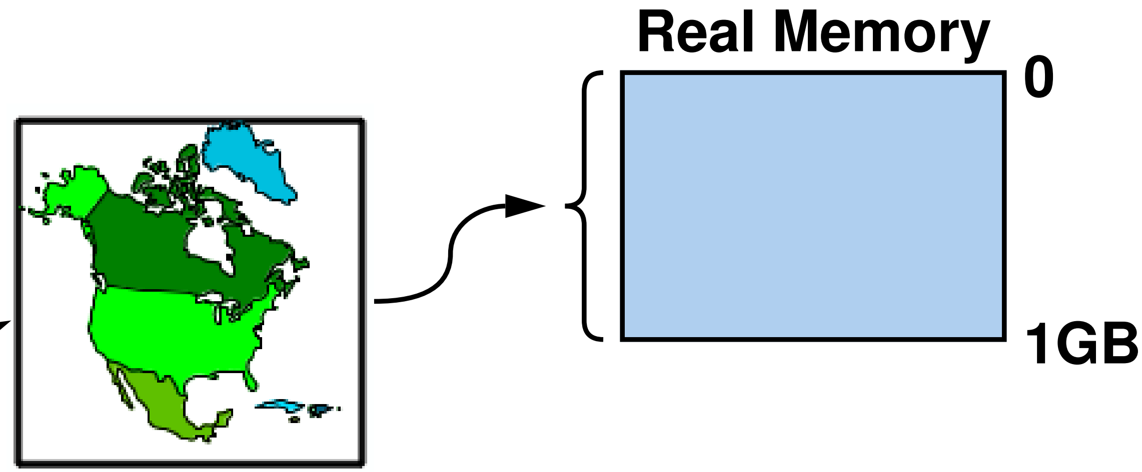




- If you only have 1GB of physical memory
  - conceptually, can setup page table so that physical address = kernel vaddr - `0xc0000000`
    - *not done this way in real systems*
  - OS can read any *physical memory location* easily

Example: 1GB Of Real Memory

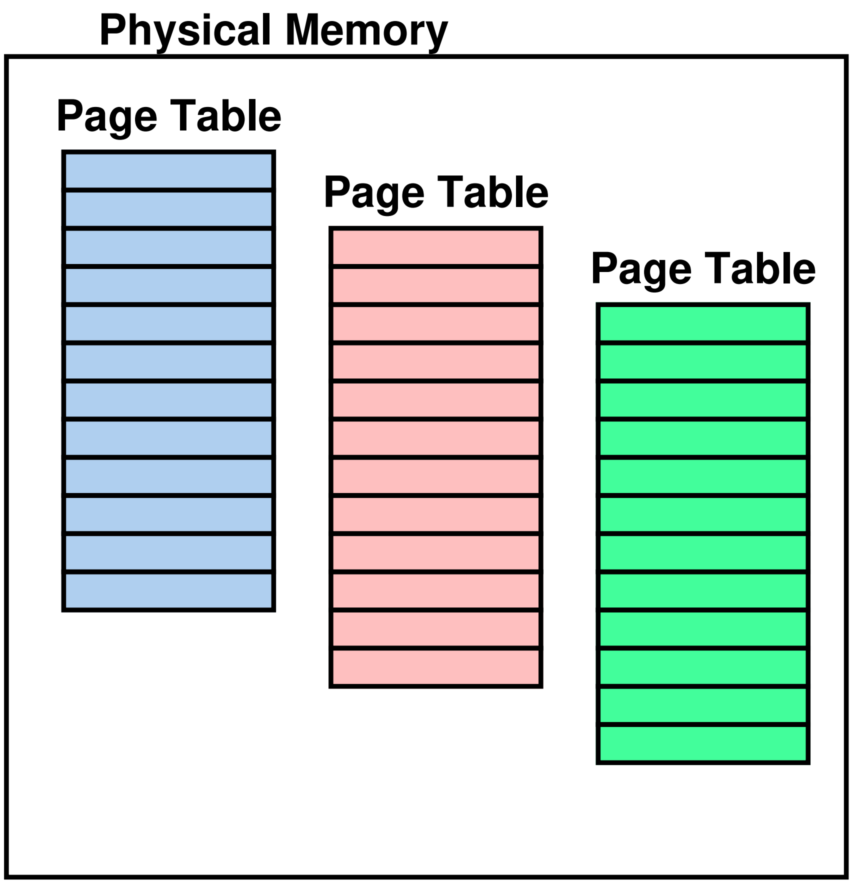


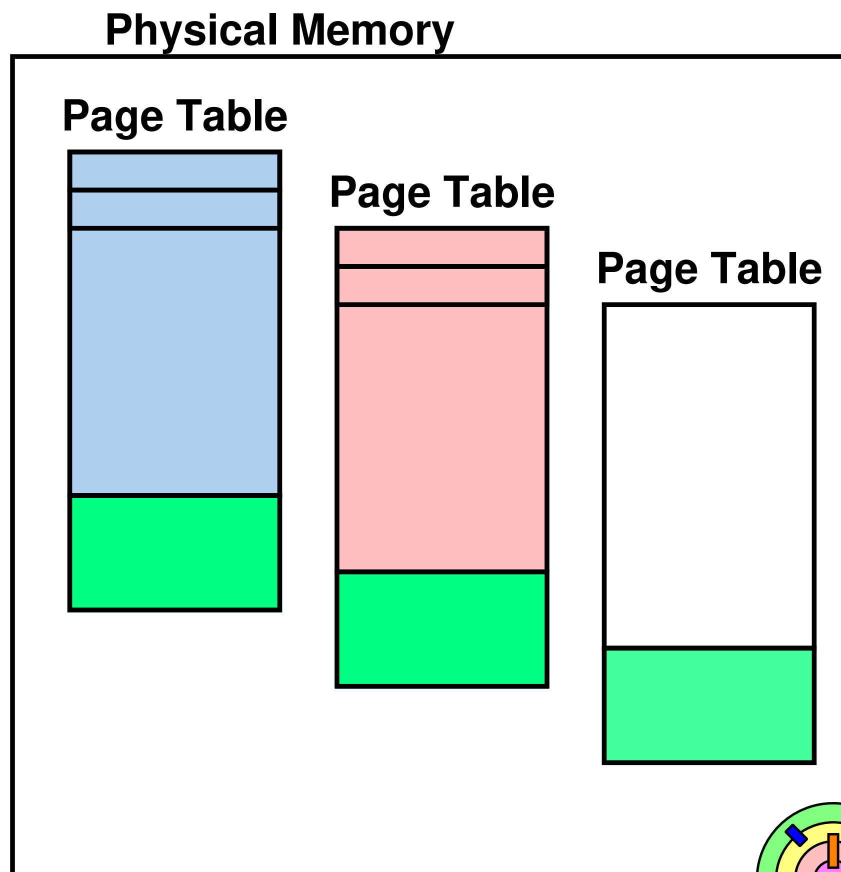


- When allocating page frames for user space memory pages
  - these pages are mapped from ***both*** user and kernel spaces
  - OS can read any ***user space memory location*** directly (assuming it's "mapped")

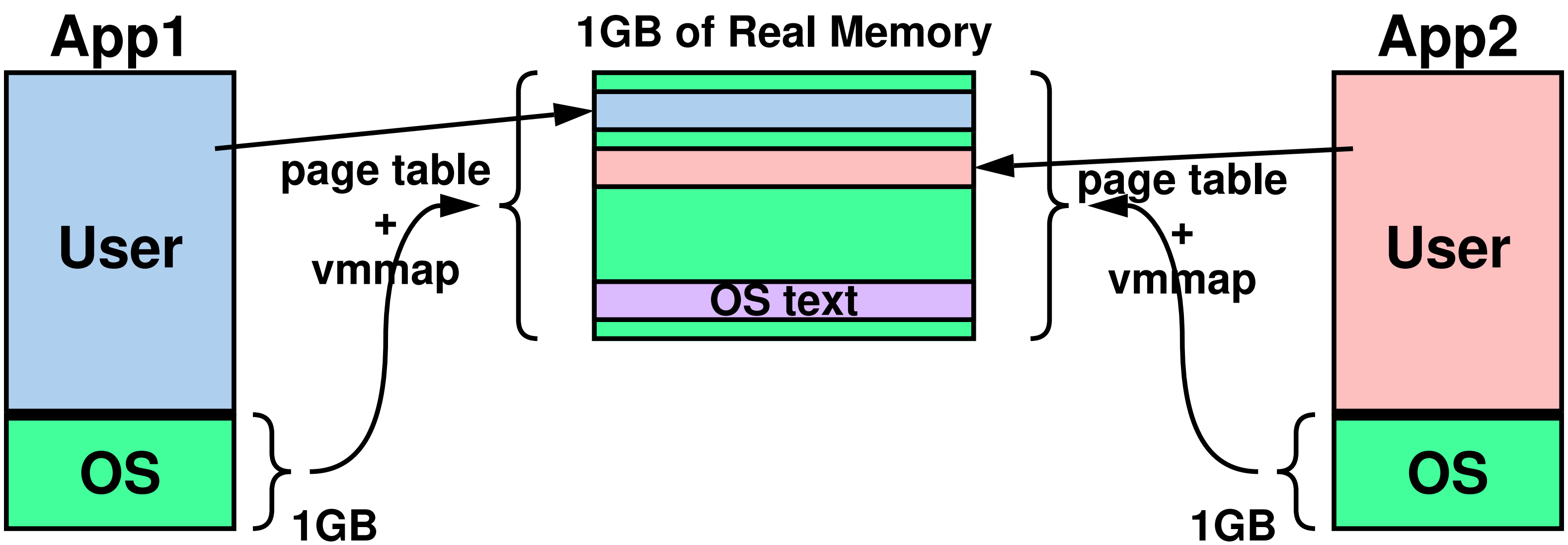
Example: 1GB Of Real Memory

- What a user thread becomes the kernel thread, it's still in the ***same process***, therefore, should use the ***same page table***
- Multiple processes - page tables
  - does ***not*** look like this
  - but look like this:



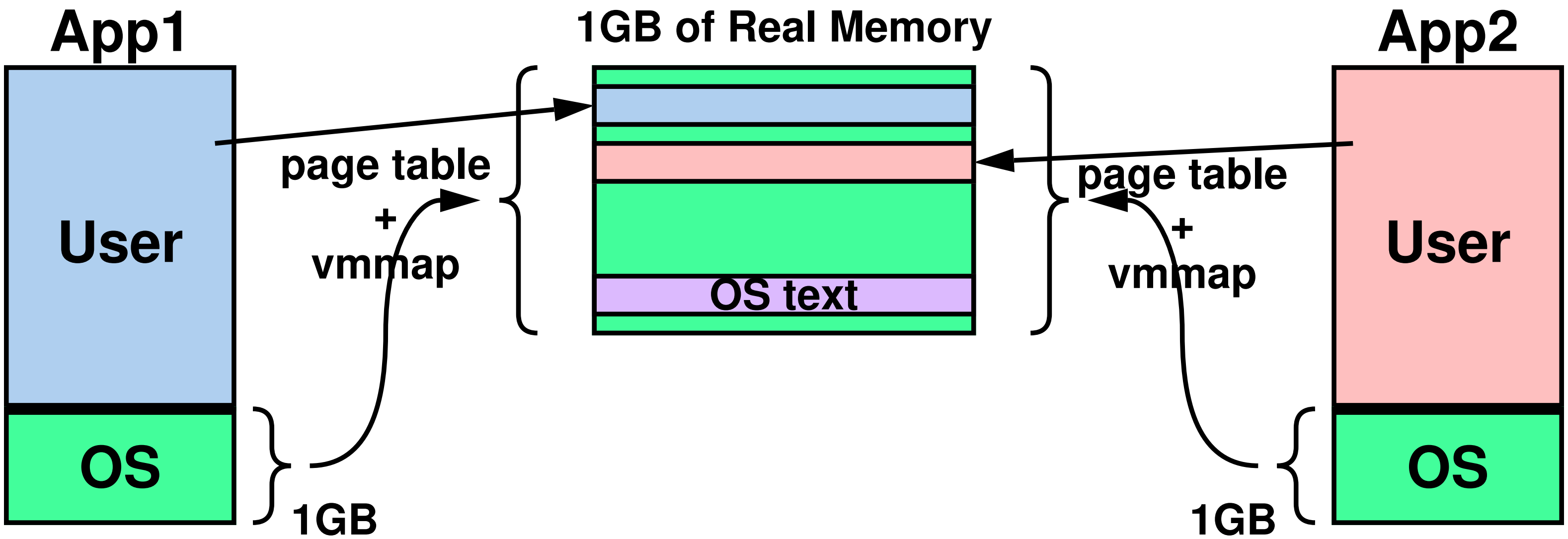


Example: 1GB Of Real Memory



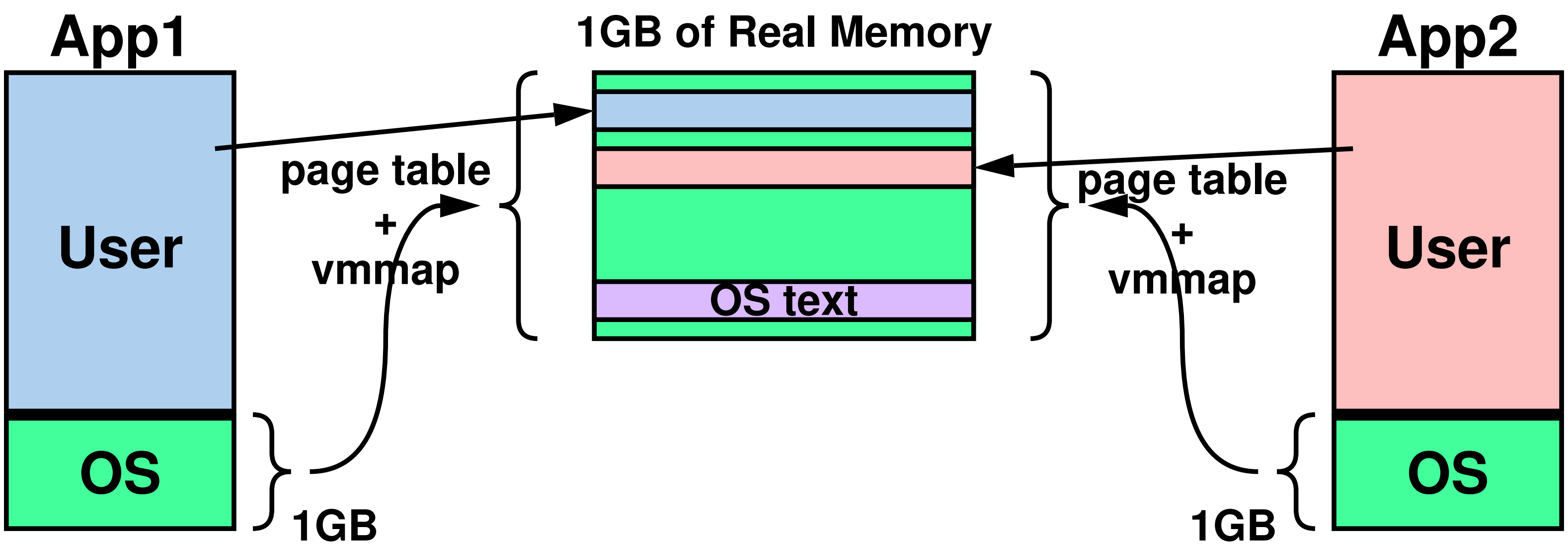
- When you switch from one process to another
  - OS code and OS data stay exactly where they were
  - bottom 1/4 of page tables of all processes are *mapped identically*
    - for kernel-only processes, only the bottom 1/4 of the page tables are mapped (i.e., top 3/4 always have V=0)
- How to setup top 3/4 page table for a user process?
  - by using the *vmmap* (virtual memory map) data structure
    - *vmmap* is only needed to manage *user portion of the address space*

Example: 1GB Of Real Memory



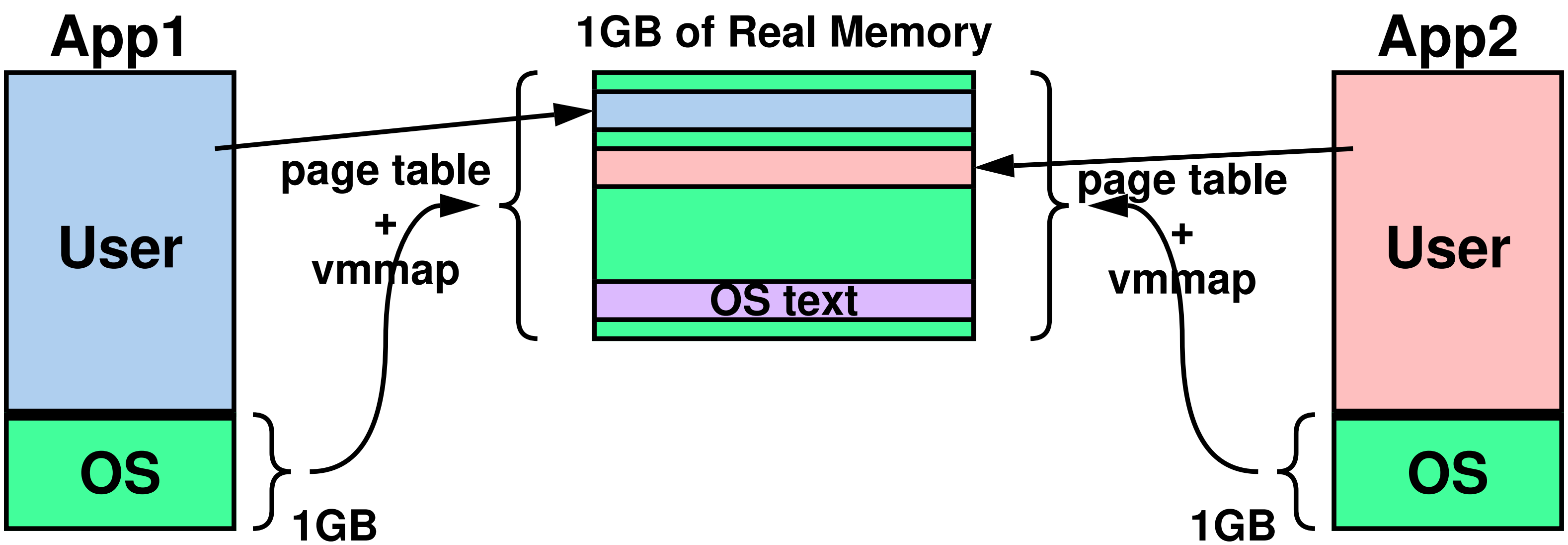
- ***Every physical address*** **that's allocated to a user process can have** ***two virtual addresses***
  - **one for the kernel and one for a user process**
  - **which virtual address should the kernel use?**
    - **be careful with user virtual address**
      - **if V bit in PTE is 0, cannot use such user virtual address in the kernel**
    - **can always use the kernel virtual address**

Example: 1GB Of Real Memory



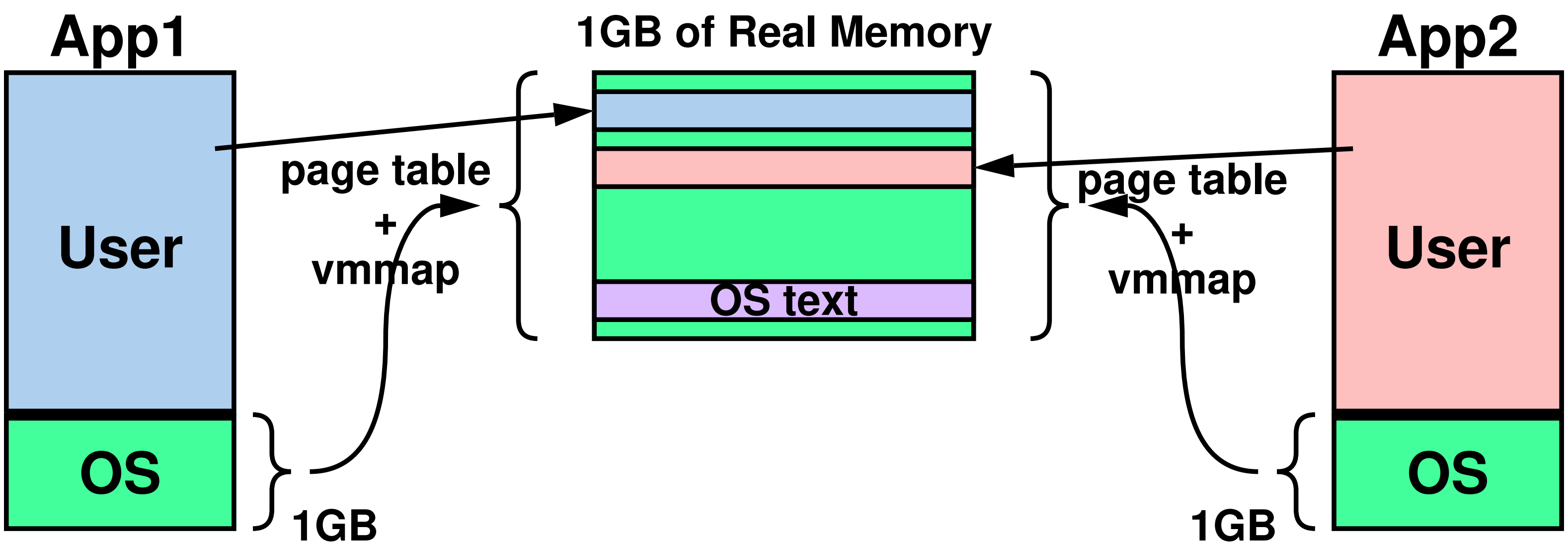
- When you trap into the kernel
  - you are still the *same thread* from *same process*
  - you use the *same page table*
    - in x86, there's a user/kernel bit in each PTE (page table entry)
      - top 3/4 of the PTEs set the bit to *U(ser)=1*
      - bottom 1/4 of the PTEs set the bit to *0 (Superviser)*
    - the *kernel part* of every page table are mapped *identically*
      - if you have 1GB or less physical memory, once this part is mapped, they will *never change*

Example: 1GB Of Real Memory



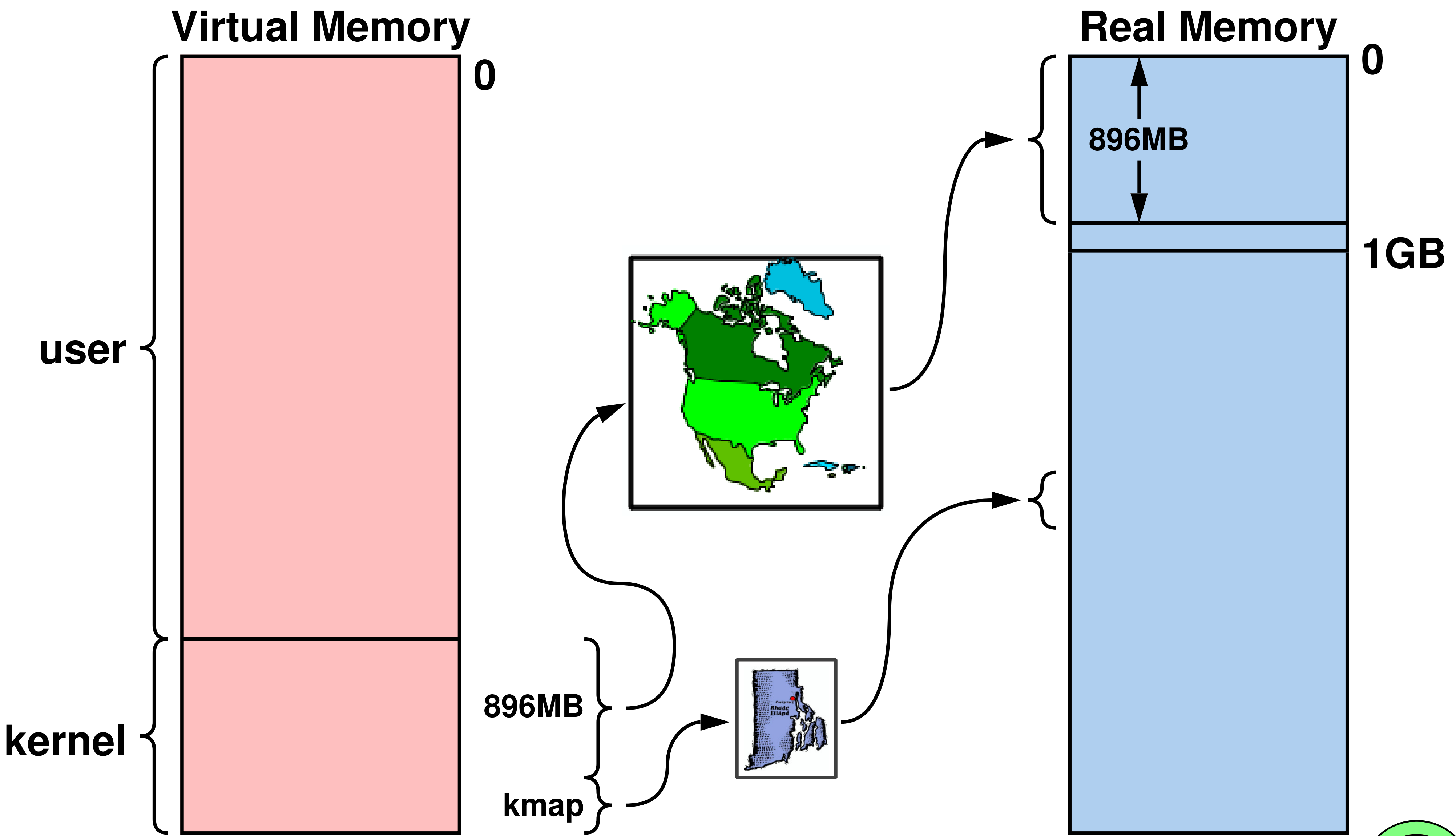
- HW: read `pt_init()` in "`kernel/mm/pagetable.c`" of `weenix`
  - kernel text, data, and bss starts at virtual address `0xc0000000`
  - then comes kernel's *page directory table* (4KB+4KB)
  - then comes kernel's *page tables* (4KB each)
  - understand how the first page table is setup for the *kernel*
  - understand that the *kernel*, just like user processes, can only use *virtual addresses!*
- Although `weenix` only has 256MB of physical memory

Example: 1GB Of Real Memory

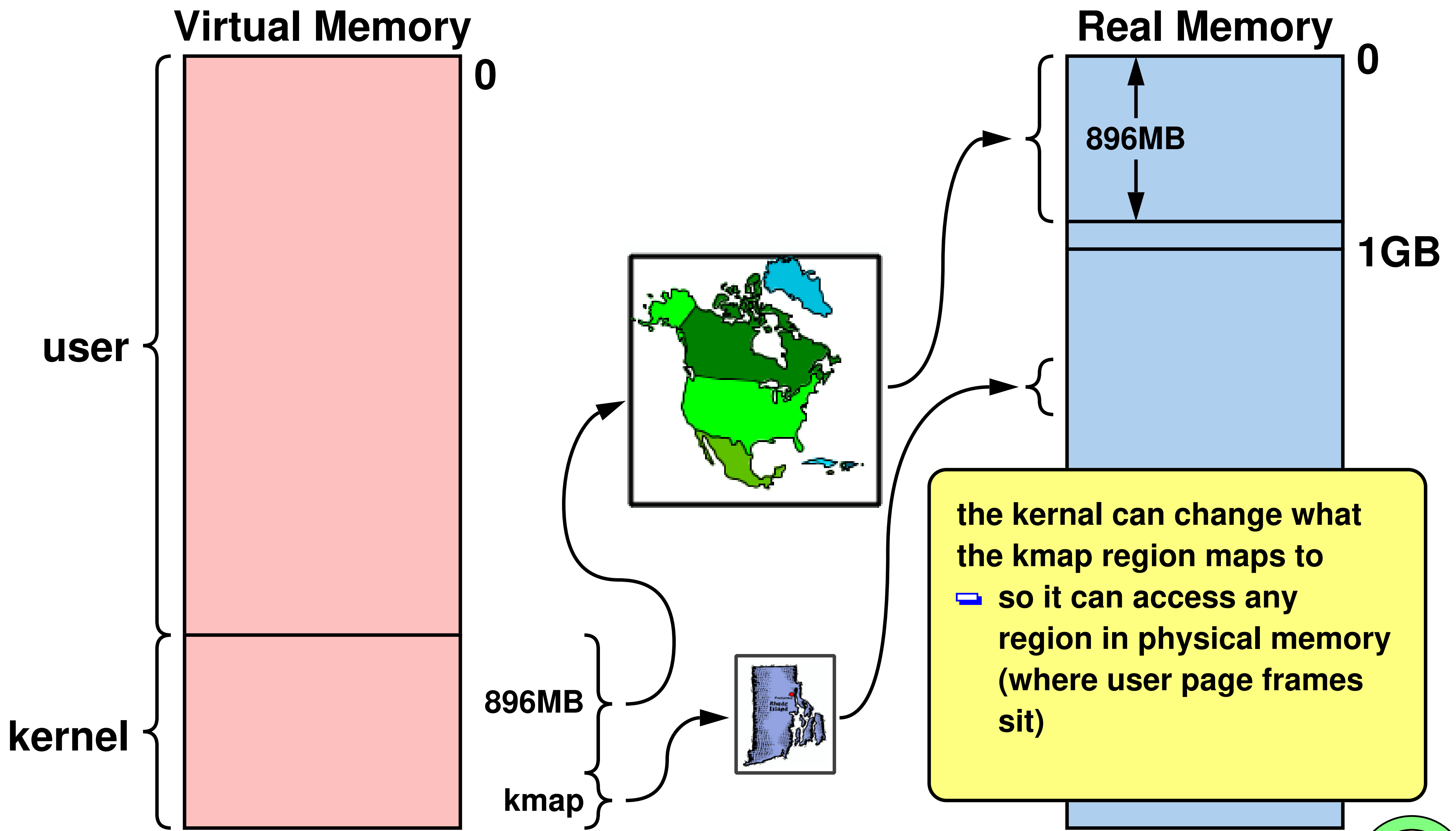


- In weenix, when an application call `read()` with buffer address `0x12345678`, how can the kernel write to this buffer?
  - should use *kernel* virtual address since it's always safe to use
  - how to convert `0x12345678` to kernel virtual address?
    - use the *vmmap* data structure
      - find memory segment it belongs (a memory segment consists of a bunch of page frames, find right page frame)
      - page frame has the base kernel virtual address (i.e., page-aligned)

Lots of Real Memory



Lots of Real Memory

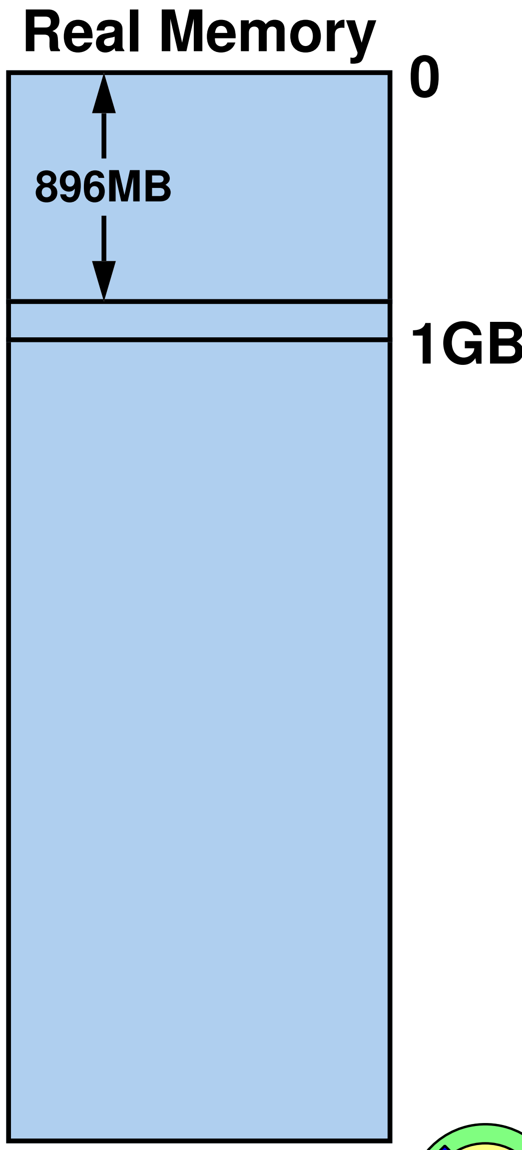


the kernal can change what the kmap region maps to

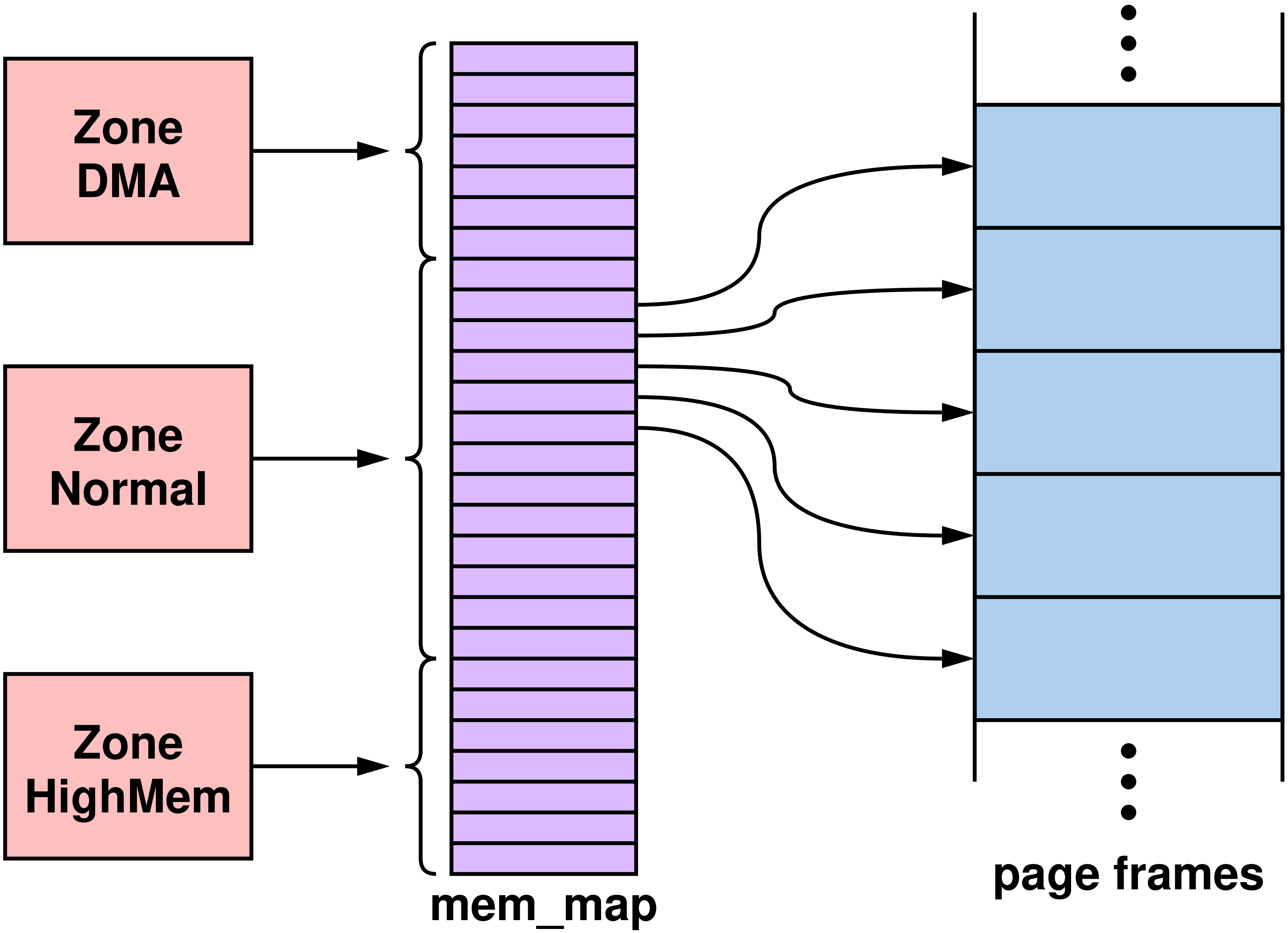
- so it can access any region in physical memory (where user page frames sit)

Mem_map and Zones

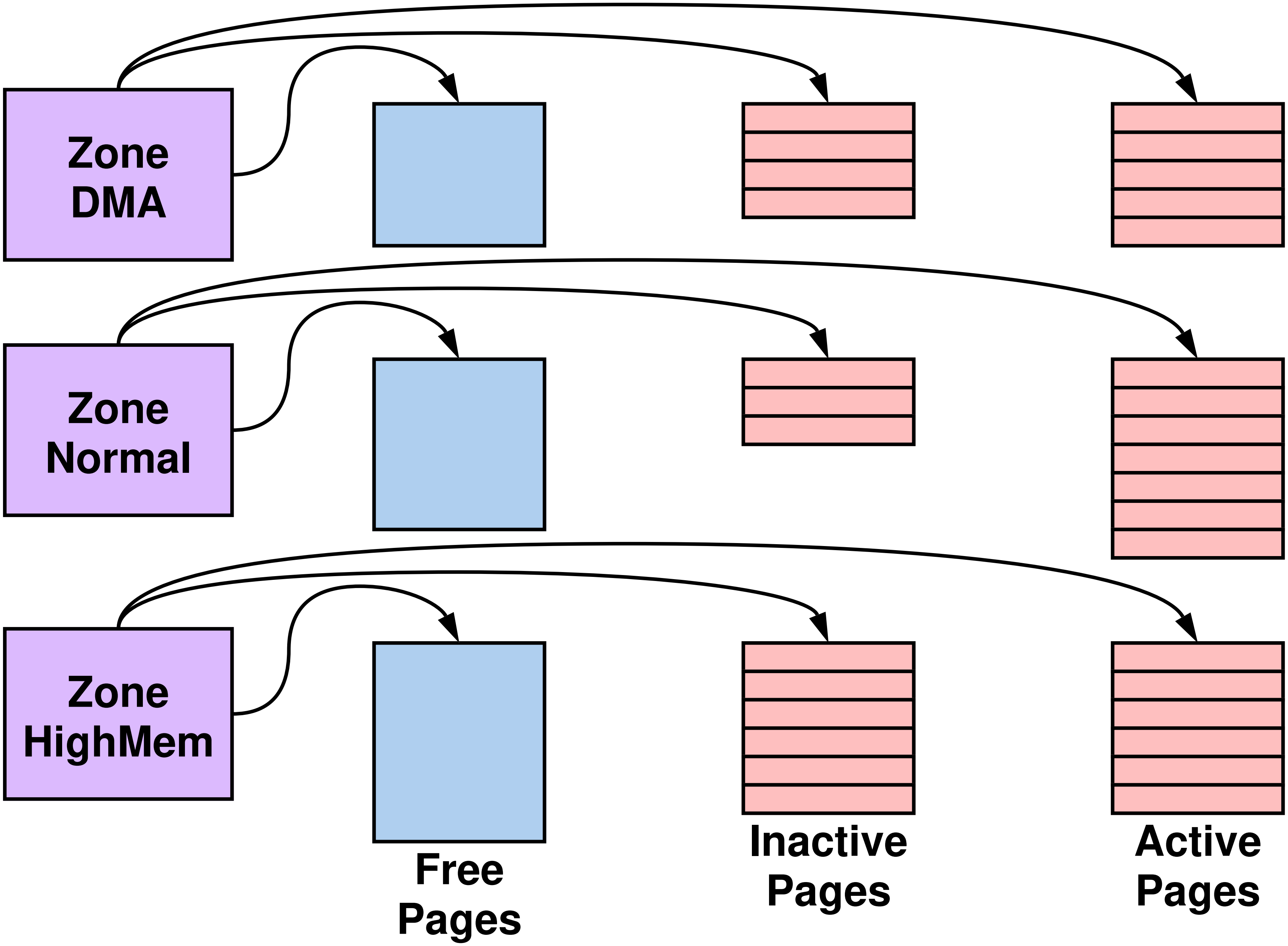
- Linux divides *physical memory* into 3 zones
  - *DMA zone:* locations $< 2^{24}$
    - `0x00000000` to `0x00ffffff`
    - many DMA devices can only handle 24-bit address
  - *Normal zone:* locations $\geq 2^{24}$ and $< 2^{30} - 2^{27}$
    - `0x01000000` to `0x37ffffff`
    - OS data structures must reside in this range
    - user pages *may* be in this range
  - *HighMem zone:* locations $\geq 2^{30} - 2^{27}$
    - `0x38000000+`
    - *strictly* for user pages



Mem_map and Zones



Page Lists

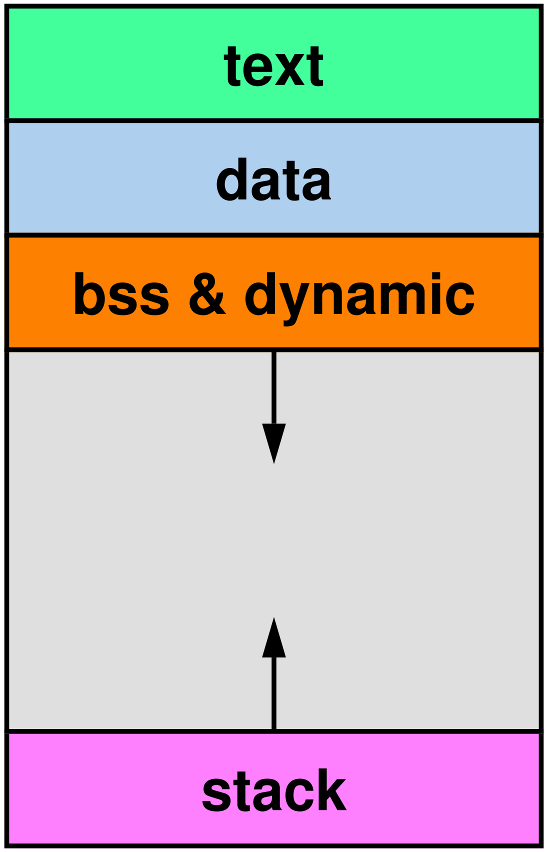


Zone DMA
Zone Normal
Zone HighMem
Free Pages
Inactive Pages
Active Pages

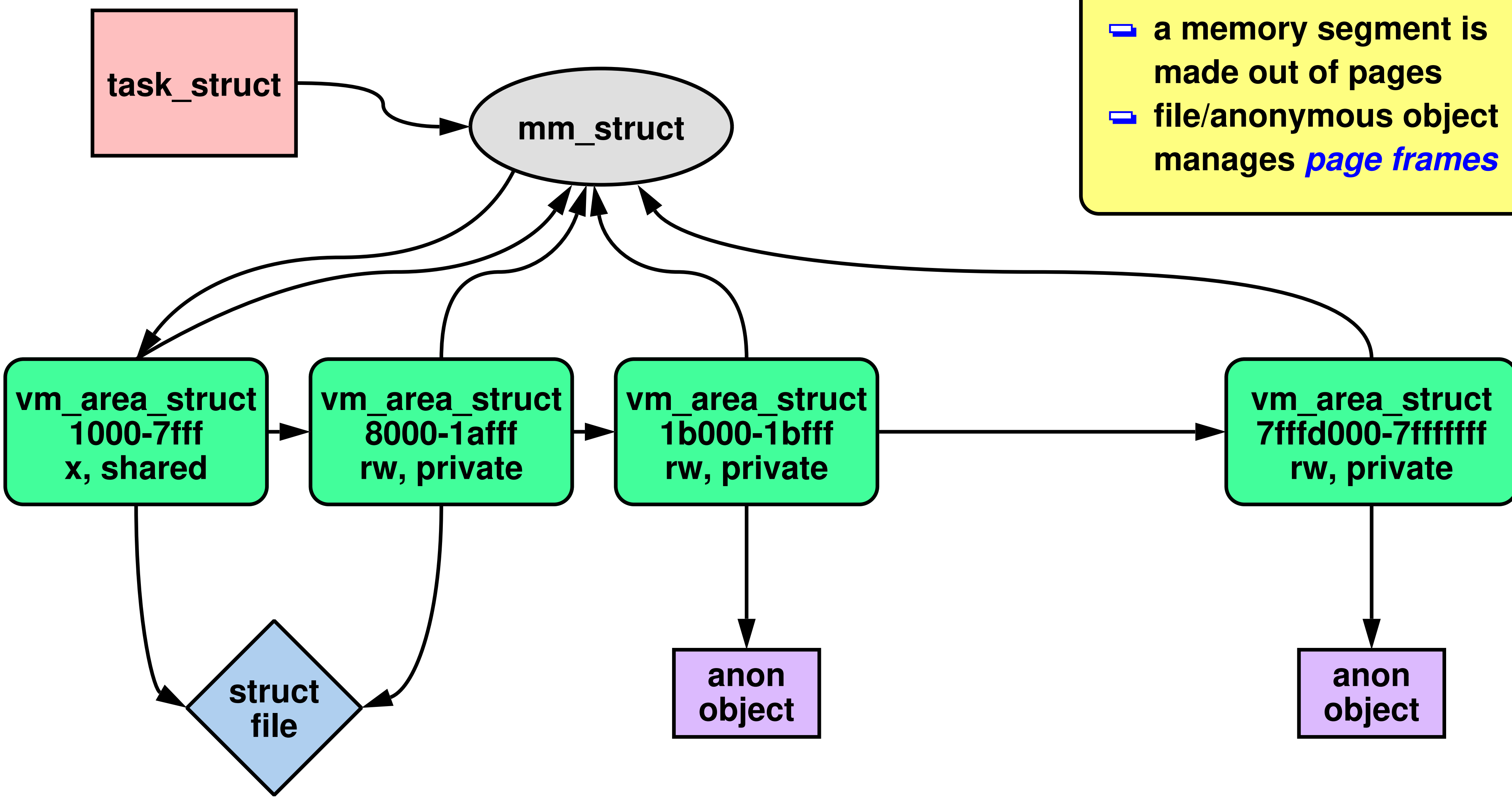
Page Lists

- Each zone's page frames are divided into three lists
  - *free list*
    - contain physical pages that have not been allocated
    - *buddy system* to maintain
  - *active*
    - picked out by *clock algorithm* as *recently used*
  - *inactive*
    - picked out by *clock algorithm* as *not recently used*
    - dirty/modified
      - marked as *"busy"* and is *unmapped* from all processes (i.e., set V=0 in PTE) that share this page
      - when you lookup a page frame in the page fault handler, if the page frame is *"busy"*, you must wait until the disk operation is finished
      - when data transfer is completed, must wake up all threads waiting for this page frame to become *"un-busy"*

Simple User Address Space



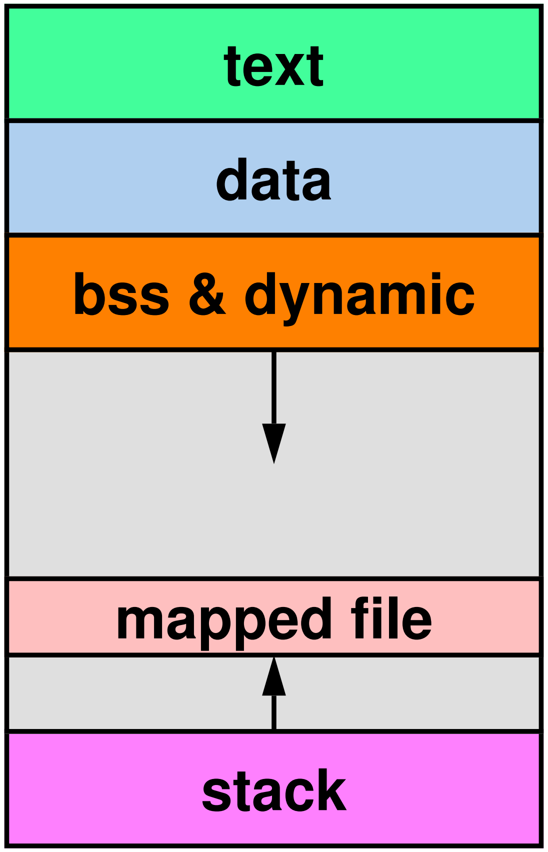
Address-Space Representation (Somewhat Simplified)



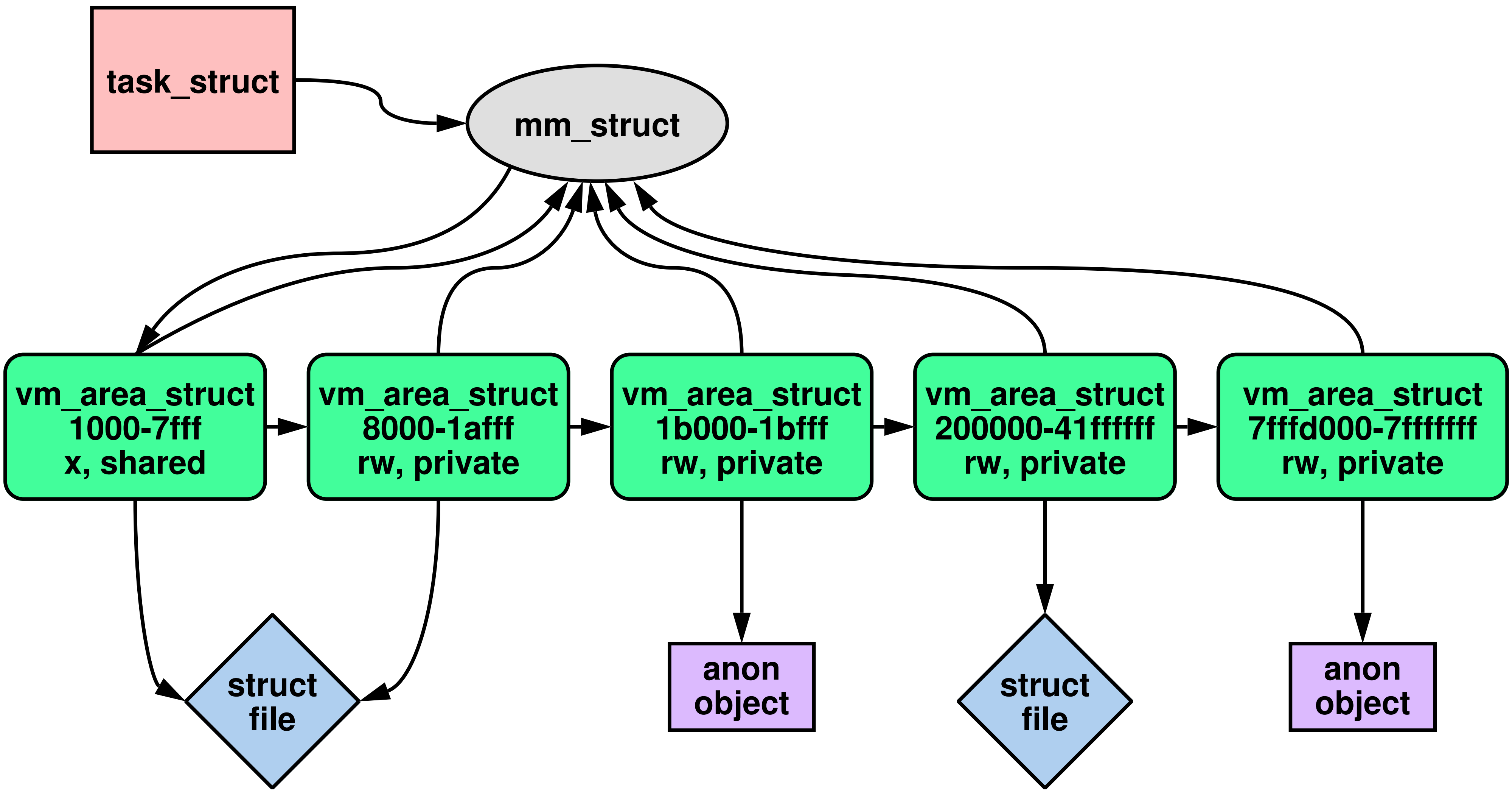
- a memory segment is made out of pages
- file/anonymous object manages *page frames*

- `vm_area_struct` is what we used to call `as_region`
  - *areas*, *regions*, *memory segments* are the same thing

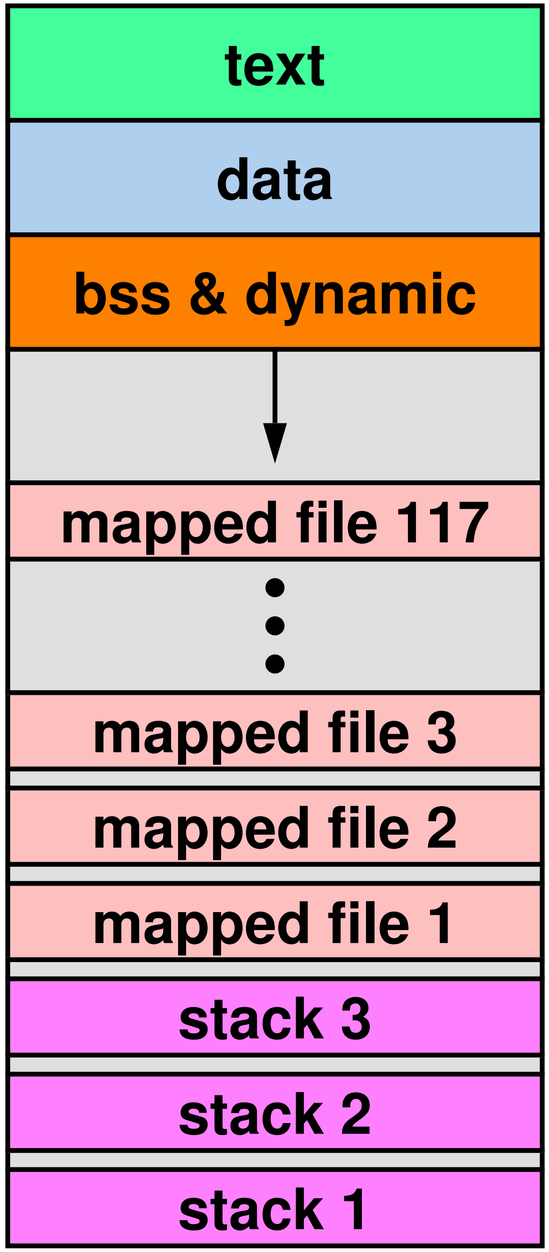
Adding a Mapped File



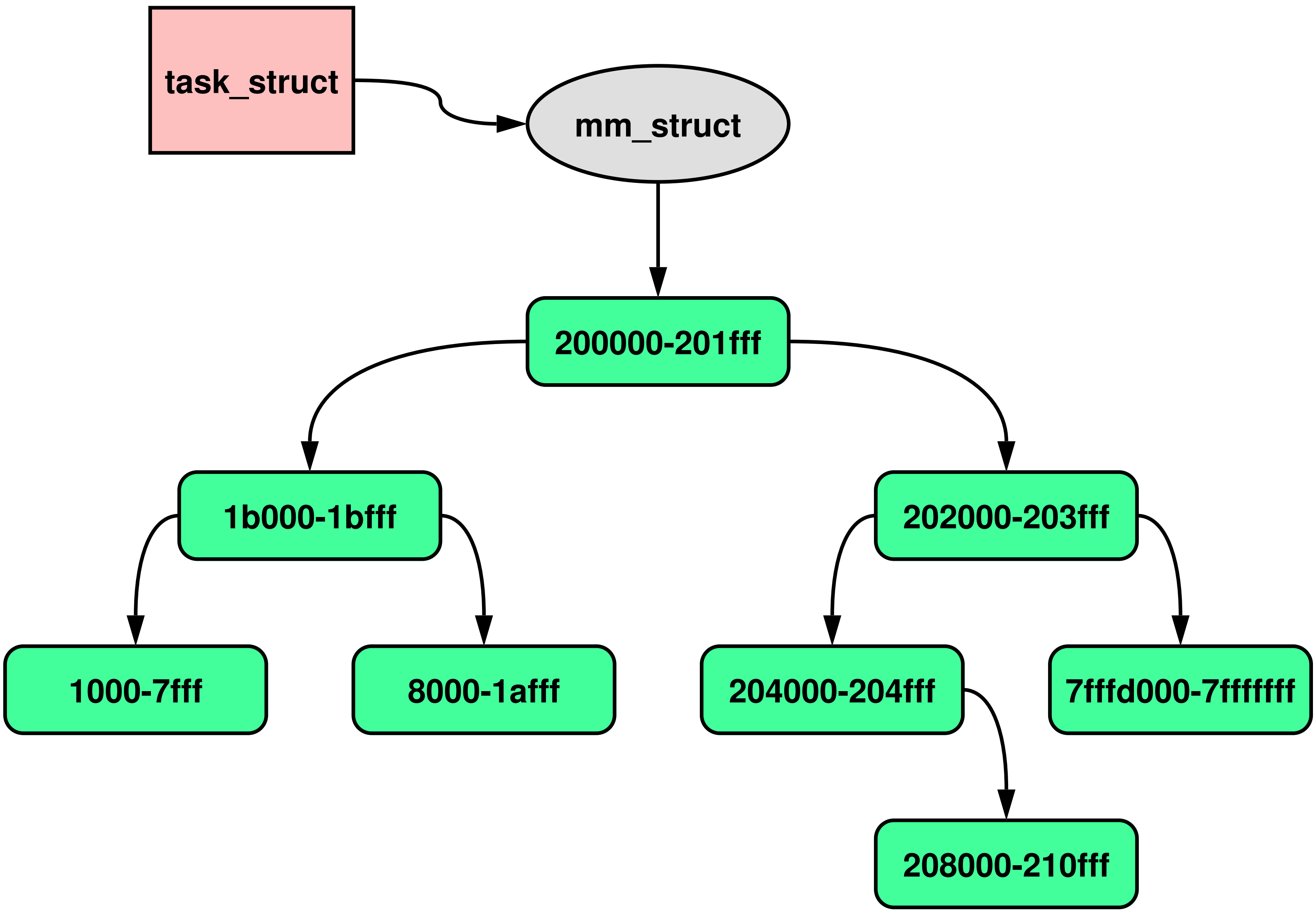
Address-Space Representation: More Areas



Adding More Stuff



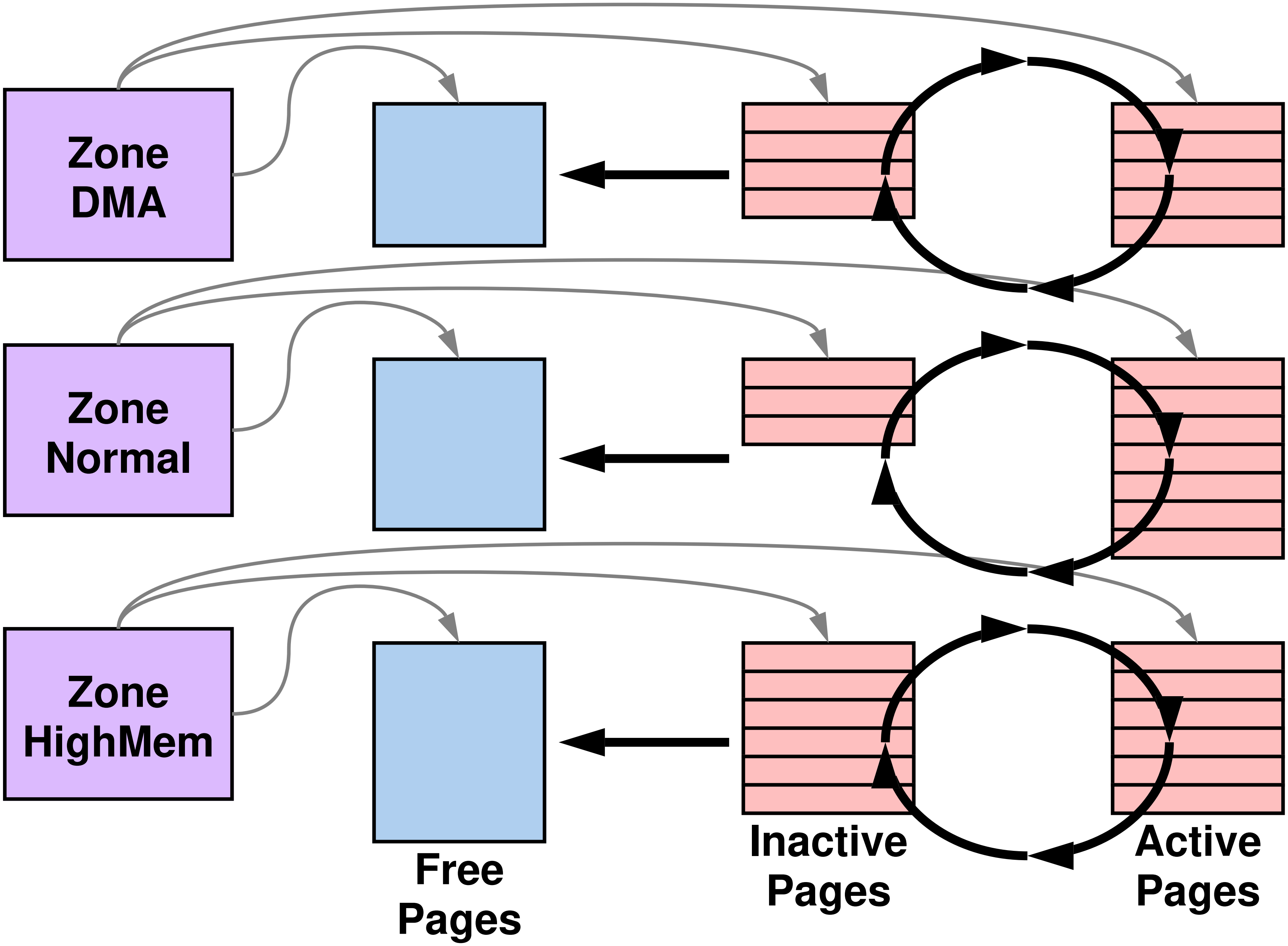
Address-Space Representation: Reality



Linux Page Management

- Replacement
  - *two-handed* clock algorithm
  - applied to zones in sequence
  - essentially *global* in scope

Page Scanning



Important Linux VM Data Structures Summary

- For each process, *PCB* contains
  - *virtual memory map* (which represents the user address space)
    - maps *virtual* memory segments
    - keeps track of *page frames* and *backing store*
  - hardware page tables
- Globally, *free, active, and inactive page list* are maintained

Important Linux VM Data Structures Summary

- For each process, *PCB* contains
  - *virtual memory map* (which represents the user address space)
    - maps *virtual* memory segments
    - keeps track of *page frames* and *backing store*
  - hardware page tables
- Globally, *free, active, and inactive page list* are maintained
- Example usage 1: What happens when a *page fault* occurs?

Important Linux VM Data Structures Summary

- For each process, *PCB* contains
  - *virtual memory map* (which represents the user address space)
    - maps *virtual* memory segments
    - keeps track of *page frames* and *backing store*
  - hardware page tables
- Globally, *free, active, and inactive page list* are maintained
- Example usage 1: What happens when a *page fault* occurs?
  1) page fault came from the hardware if V=0 for a page
  2) traps into the kernel, the kernel:
     - 2a) gets a free page frame
     - 2b) looks at the *virtual memory map* and copy the page from disk into this free page frame
     - 2c) adjust hardware page table to point to this page frame

Important Linux VM Data Structures Summary

- For each process, *PCB* contains
  - *virtual memory map* (which represents the user address space)
    - maps *virtual* memory segments
    - keeps track of *page frames* and *backing store*
  - hardware page tables
- Globally, *free, active, and inactive page list* are maintained
- Example usage 1: What happens when a *page fault* occurs?
  1) page fault came from the hardware if V=0 for a page
  2) traps into the kernel, the kernel:
     2a) gets a free page frame
     2b) looks at the *virtual memory map* and copy the page from disk into this free page frame
     2c) adjust hardware page table to point to this page frame
  - can get complicated because a page frame may be shared by multiple user processes

Important Linux VM Data Structures Summary

- For each process, *PCB* contains
  - *virtual memory map* (which represents the user address space)
    - maps *virtual* memory segments
    - keeps track of *page frames* and *backing store*
  - hardware page tables
- Globally, *free, active, and inactive page list* are maintained
- Example usage 2: What happens when *pageout daemon* wants to free up a *modified/dirty* page?

Important Linux VM Data Structures Summary

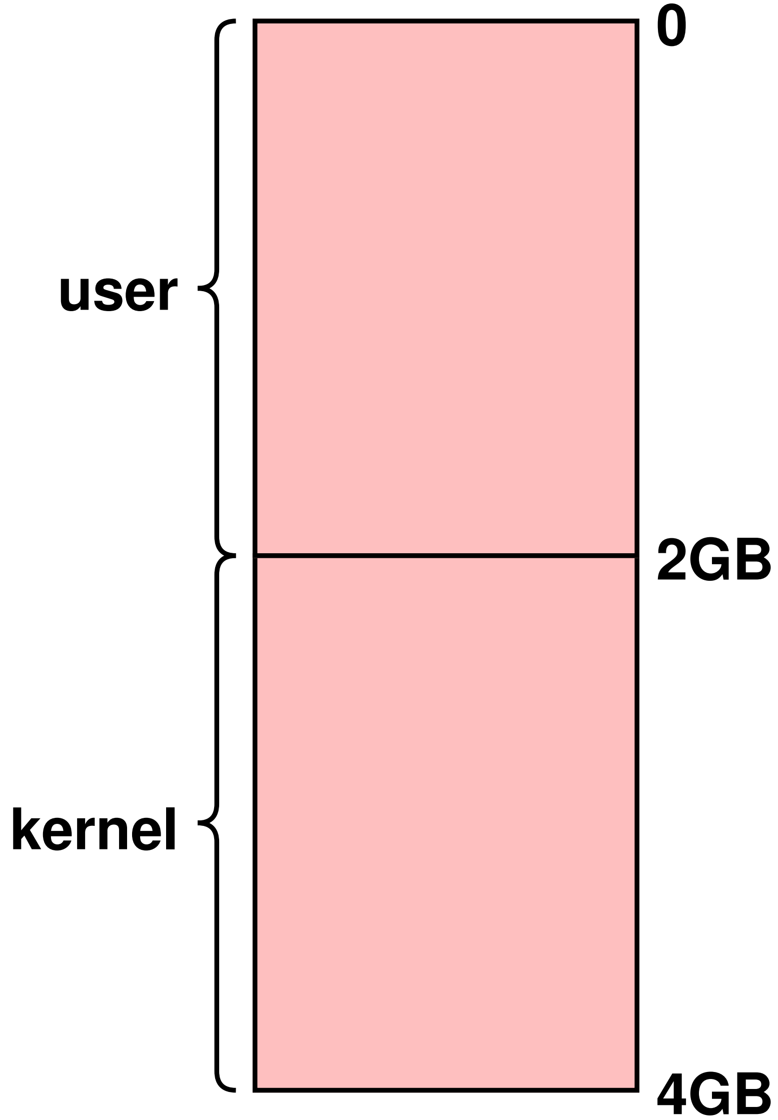
- For each process, *PCB* contains
  - *virtual memory map* (which represents the user address space)
    - maps *virtual* memory segments
    - keeps track of *page frames* and *backing store*
  - hardware page tables

- Globally, *free, active, and inactive page list* are maintained

- Example usage 2: What happens when *pageout daemon* wants to free up a *modified/dirty* page?
  1) find from which processes/address spaces the page frame belongs to
  2) *unmap* this page from the corresponding *pagetables*
     - read `pframe_remove_from_pts()` in `weenix`
  3) find the corresponding backing store, write back the page content to disk (mark the page frame "busy" while writing)
  4) free the page frame if no process is waiting to use it

Windows x86 Layout

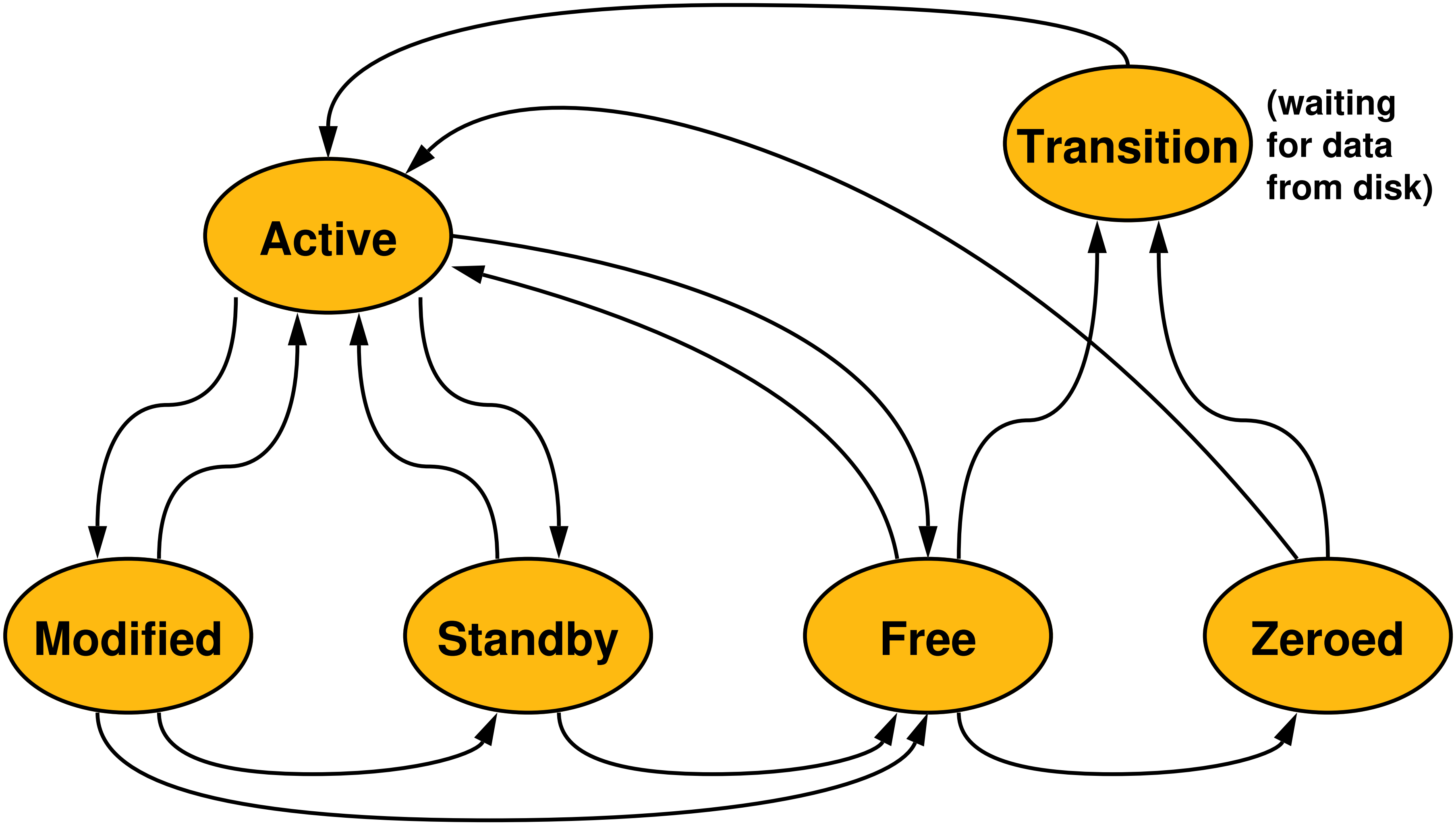
- Two choices



Windows Paging Strategy Highlights

- All processes guaranteed a "working set"
  - lower bound on page frames
  - you can get "cannot start a process because there is not enough memory" message
- Competition for additional page frames
- "Balance-set" manager thread maintains working sets
  - one-handed clock algorithm
- *Swapper* thread swaps out idle processes (inactive for 15 seconds)
  - first kernel stacks
  - then working set
  - very different from Linux
- Some of kernel memory is *paged*
  - page faults are possible
    - makes more physical memory available
    - must "*lock down*" page frames for page fault handler

Windows Page-Frame States



7.3 Operating System Issues

- General Concerns
- Representative Systems
- *Copy on Write and Fork*
- Backing Store Issues

Unix and Virtual Memory: The `fork()`/`exec()` Problem

- **Naive implementation:**
  - `fork()` actually makes a copy of the parent's address space for the child
  - child executes a few instructions (setting up file descriptors, etc.)
  - child calls `exec()`
  - result: a lot of time wasted copying the address space, though very little of the copy is actually used

vfork()

- Don't make a copy of the address space for the child; instead, give the address space to the child
  - the parent is suspended until the child returns it
- The child executes a few instructions, then does an exec
  - as part of the exec, the address space is handed back to the parent
- Advantages
  - very efficient
- Disadvantages
  - works only if child does an exec
  - child must not intentionally or accidentically modify the address space

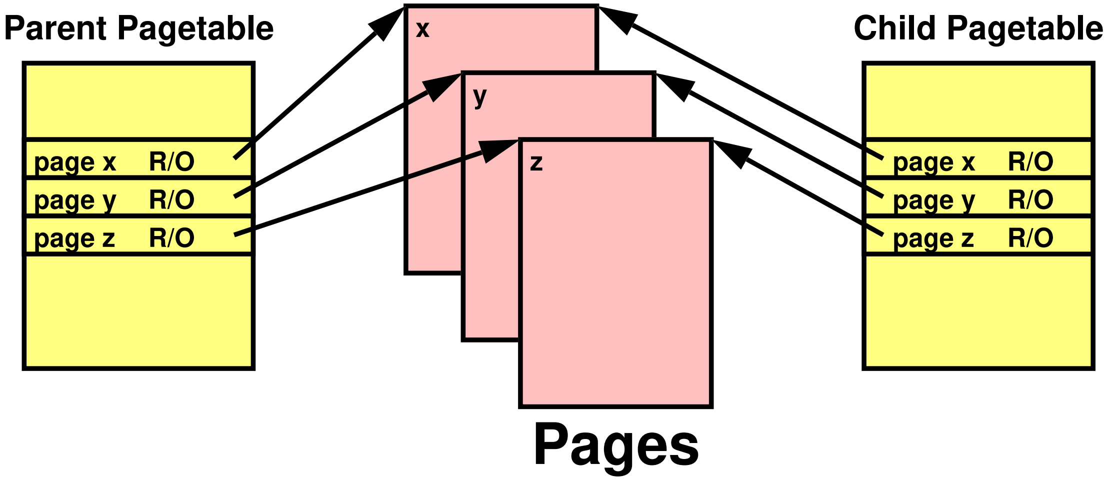
A Better `fork()`

- Parent and child share the pages comprising their address spaces
  - if either party attempts to modify a page, the modifying process gets a copy of just that page
- Principle of *Lazy Evaluation* at work
  - try to put things off as long as possible if you don't have to do them now
    - if it needs to be done now, you don't really have a choice
  - if you wait long enough, it might turn out that you don't have to do them at all
- Advantages
  - semantically equivalent to the original `fork()`
  - usually faster than the original `fork()`
- Disadvantages
  - slower than `vfork()`

Copy on Write and `fork()`

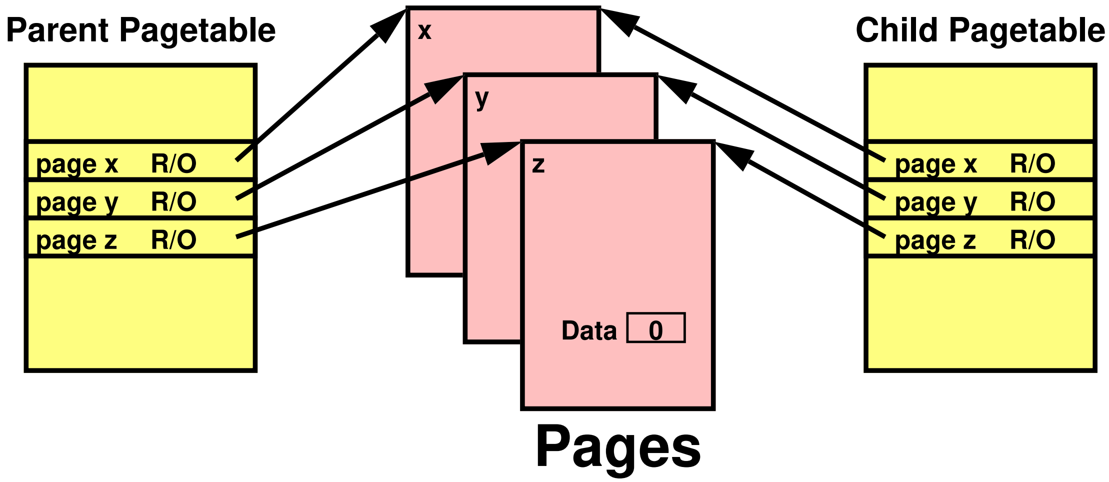
- To implement the "better" `fork()`, we need to use *copy-on-write*
  - a process gets a *private* copy of a page after a thread in the process performs a *write* to that page for the *first time*
    - set every PTE to *R/O* for pages that correspond to memory segments that needs *copy-on-write* (i.e., privately mapped)
    - during page fault, if a virtual memory segment is *R/W* and *privately mapped*, then we need to perform *copy-on-write*
      - *make a copy* of that page, set corresponding PTE to *R/W* and change its physical page number to point to the copy
  - *copy-on-write* must work with `fork()`
    - what are the complications?

Private Mapping - Copy on Write Occurs after `fork()`



- Parent and child process share pages, all marked *read-only* at first
  - to initalize the child's page table, just use `memcpy()` to copy the entire page table from the parent

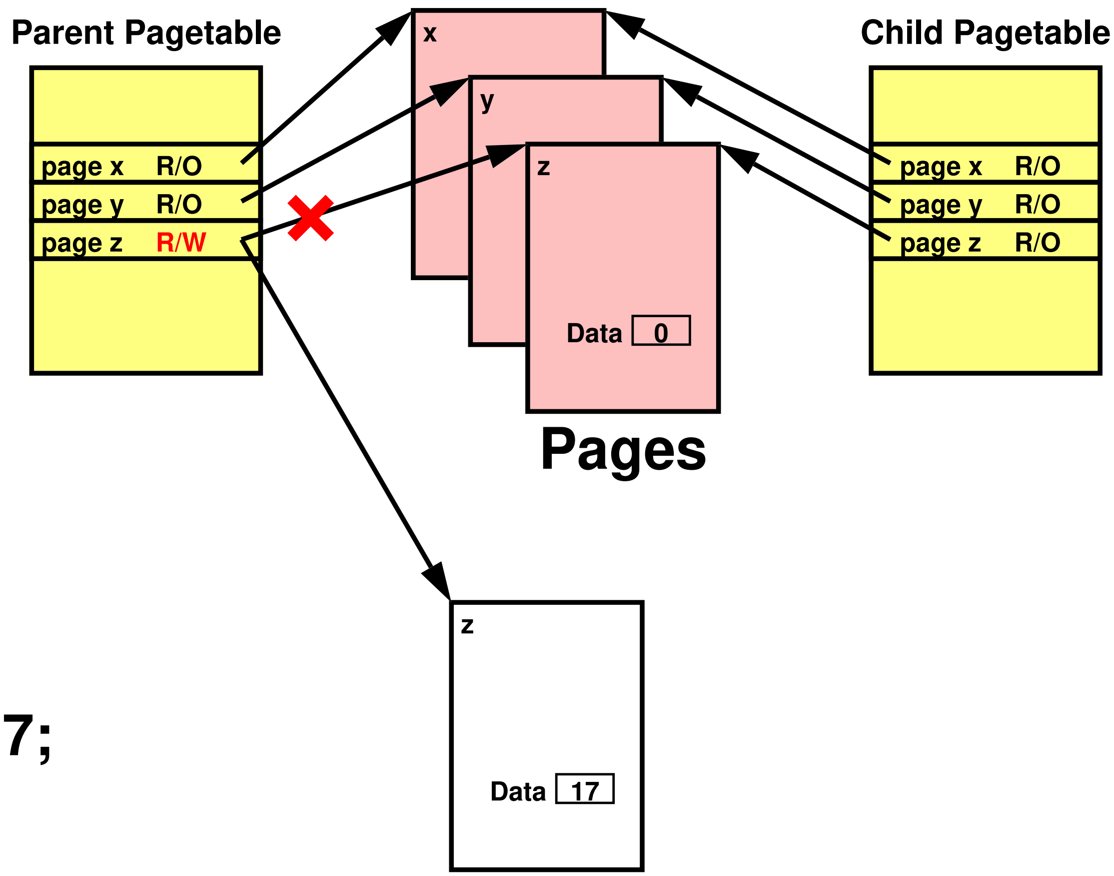
Private Mapping - Copy on Write Occurs after `fork()`



Data = 17;

- **Parent and child process share pages, all marked *read-only* at first**
  - ***copy on write:*** **when one of the processes tries to modify the data, a copy of the page is created and used**
    - **this is another reason for a *page fault***

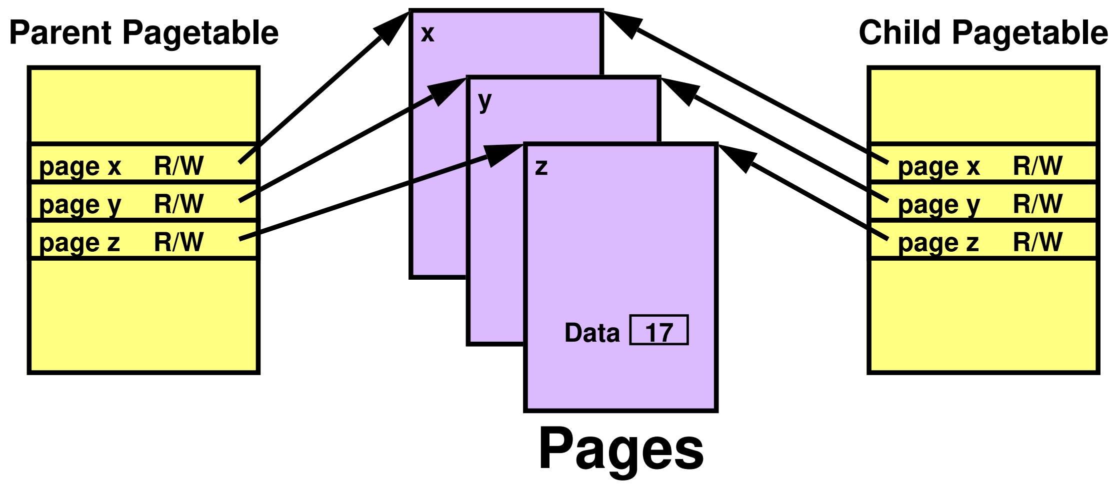
Private Mapping - Copy on Write Occurs after `fork()`



Data = 17;

- Parent and child process share pages, all marked *read-only* at first
  - *copy on write:* when one of the processes tries to modify the data, a copy of the page is created and used
    - this is another reason for a *page fault*

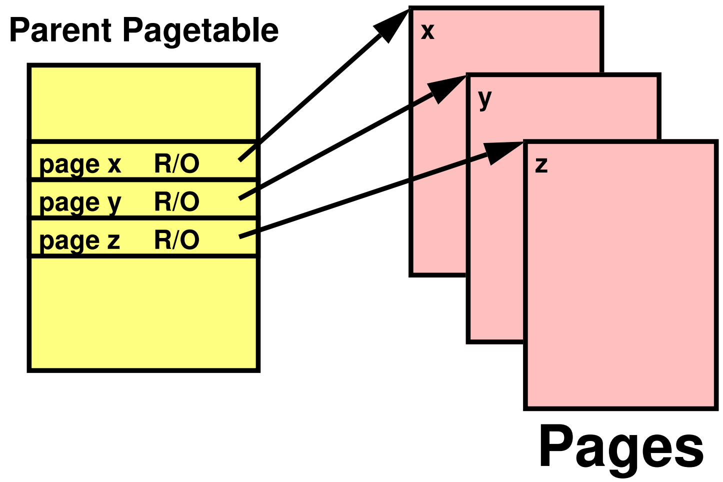
Share-Mapped Files



Data = 17;

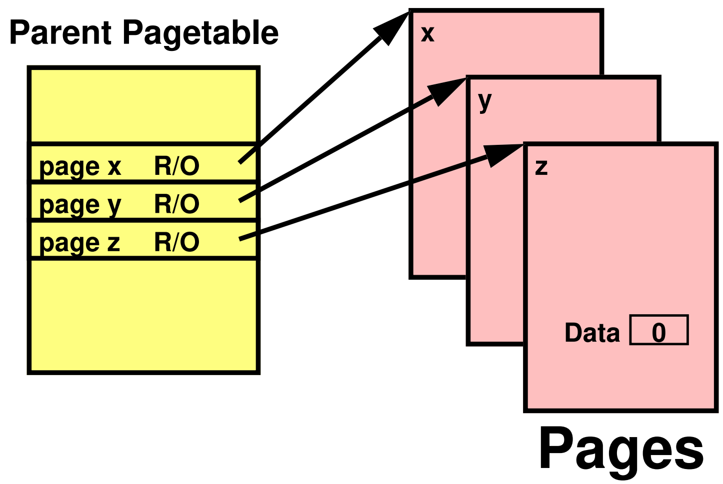
- For *shared* mapping, changes are writting into the shared page
  - please note that the information about whether a page is *shared* or *private* is *not* inside the page table
    - it is kept in a kernel data structure (`vm_area_struct`)

Private Mapping - Copy on Write before `fork()`



- For *private* mapping, *copy on write*

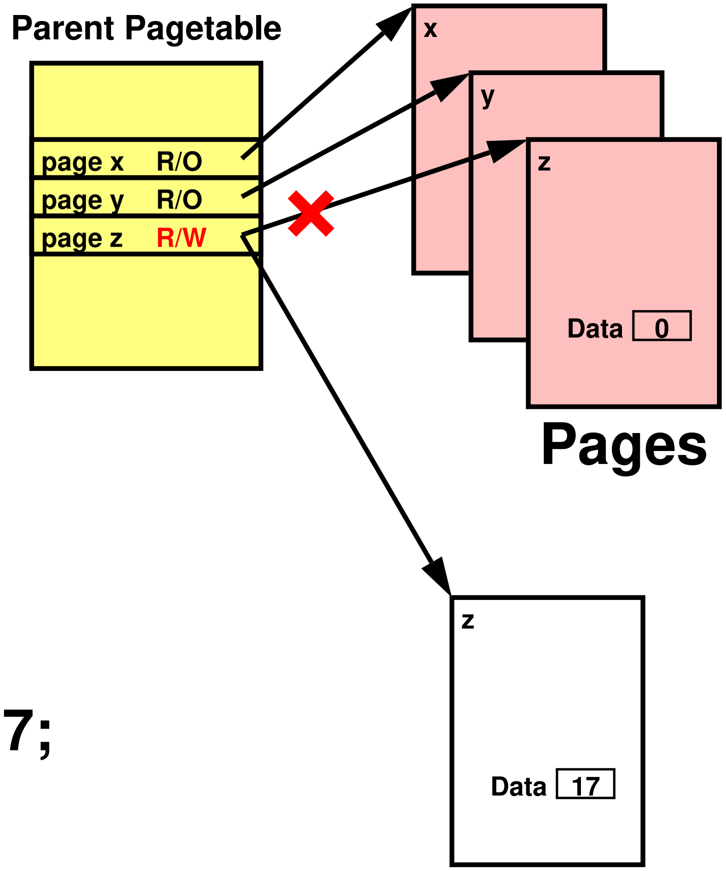
Private Mapping - Copy on Write before `fork()`



Data = 17;

For *private* mapping, *copy on write*

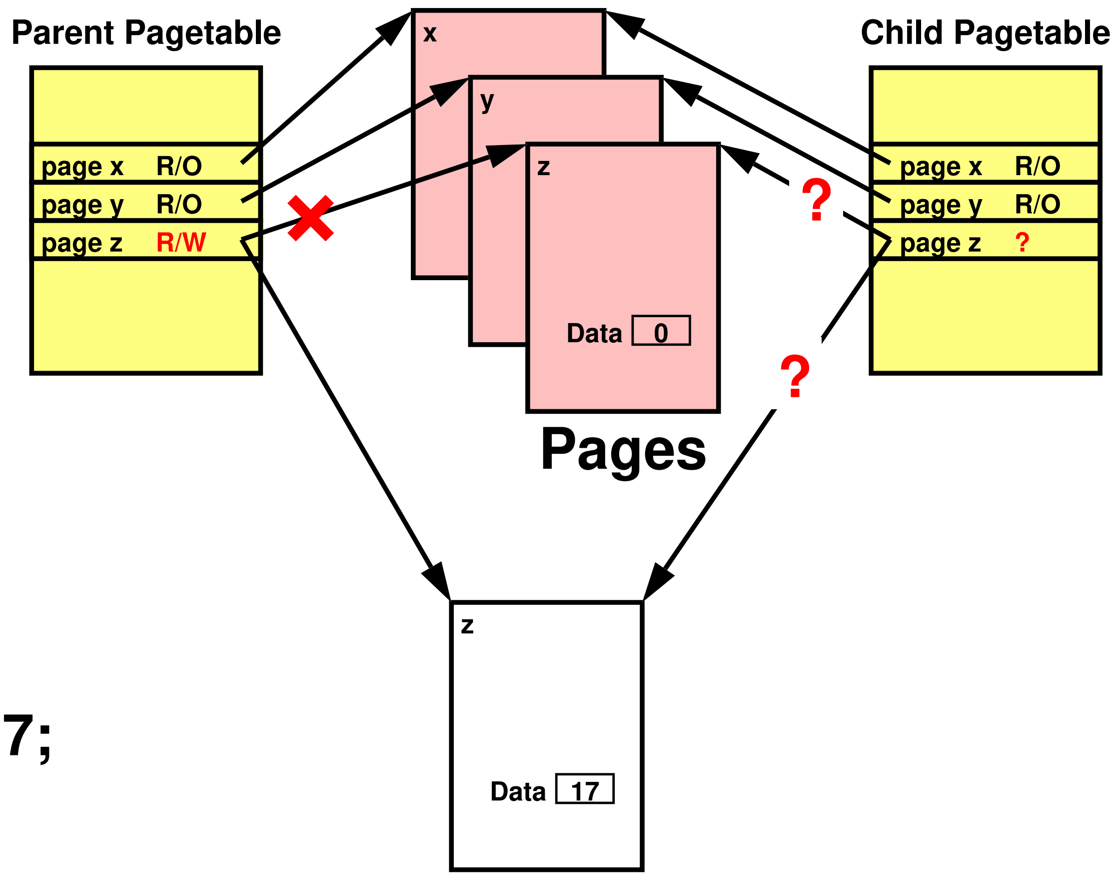
Private Mapping - Copy on Write before `fork()`



Data = 17;

➡ **For *private* mapping, *copy on write***

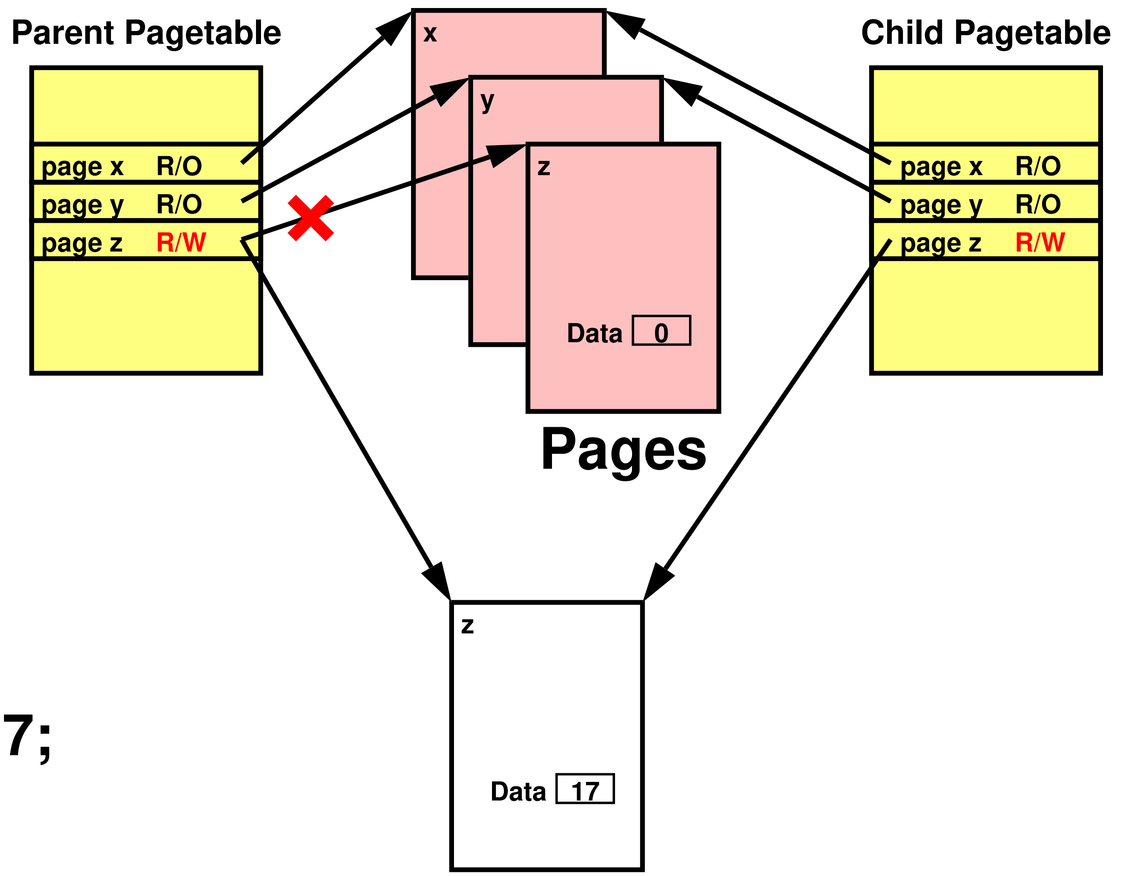
A Private-Mapped File Changes



Data = 17;

- Complication: what if the page is modified before `fork()`?
  - should child process' page be marked "modified"?
    - some of child's pages are initialized from files and some are initialized from the parent's address space

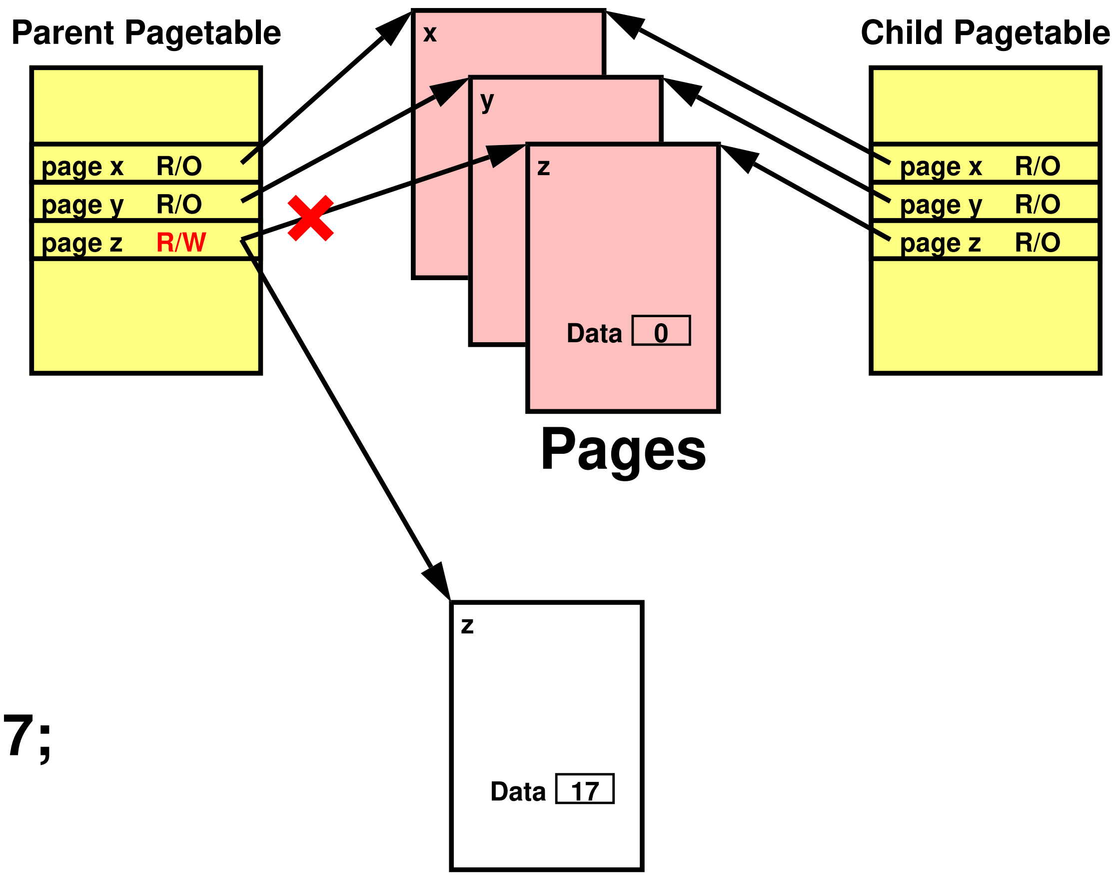
A Private-Mapped File Changes



Data = 17;

- Complication: what if the page is modified before `fork()`?
  - `memcpy()` the parent's page table is wrong: what if the parent modify the page further?
    - child should not see these changes

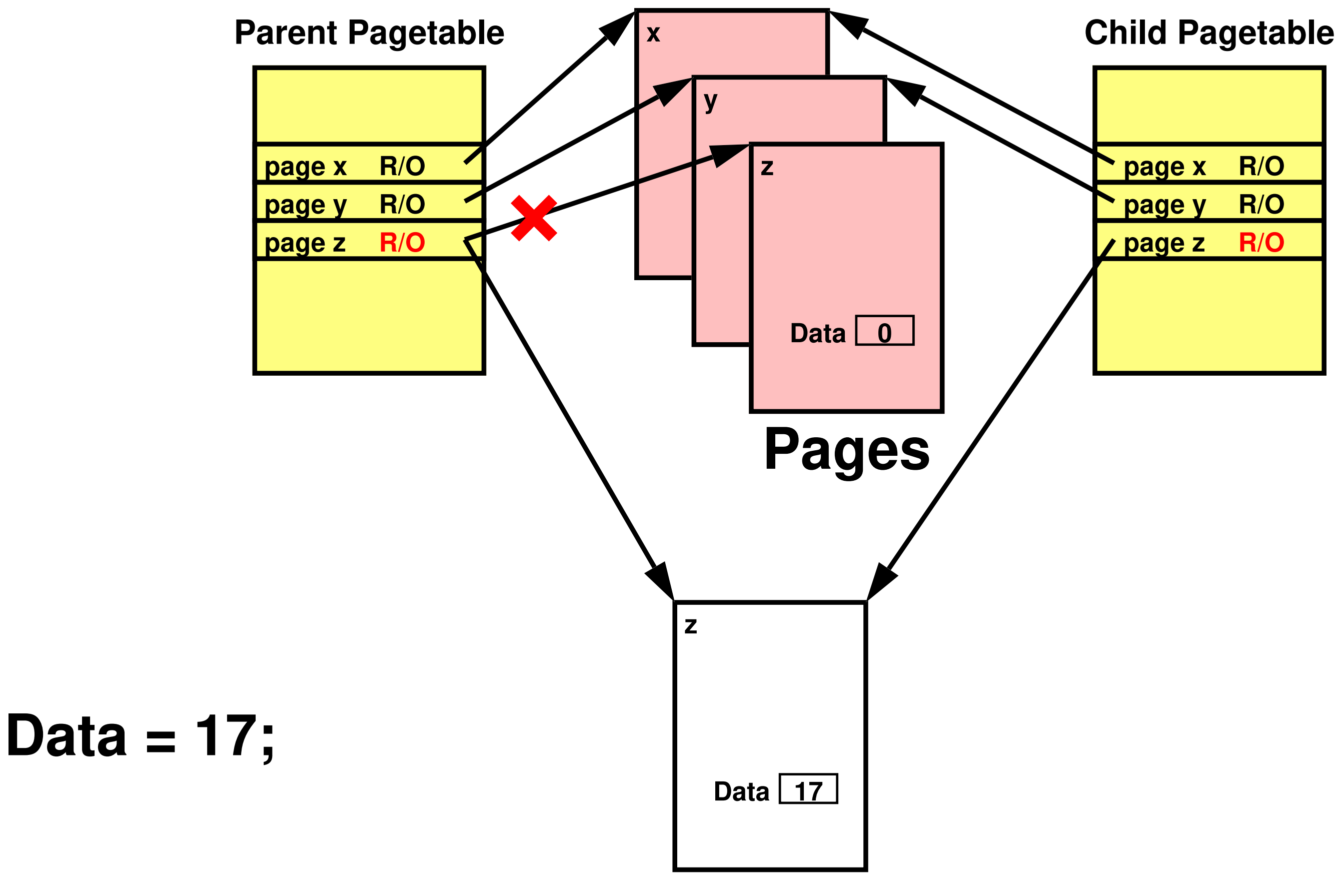
A Private-Mapped File Changes



Data = 17;

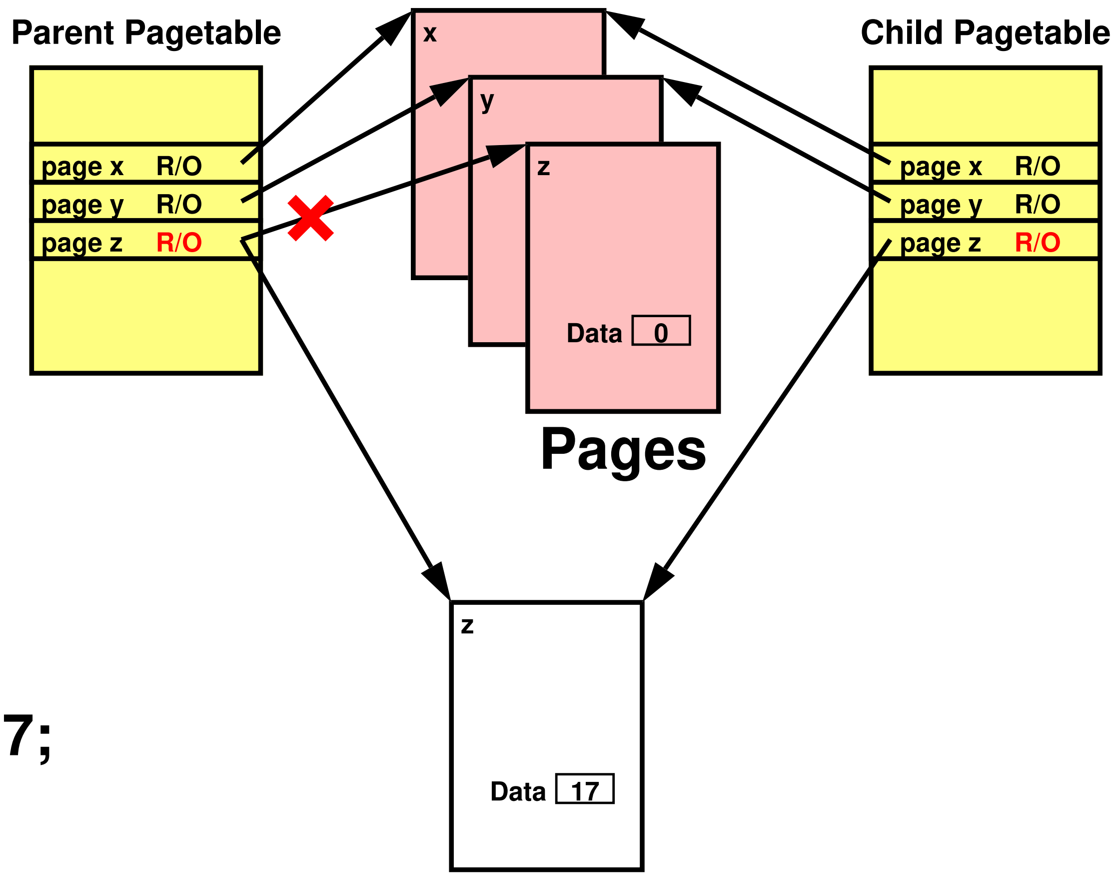
- Complication: what if the page is modified before `fork()`?
  - this is also wrong
    - child process should see 17 in Data on page z

A Private-Mapped File Changes



- Complication: what if the page is modified before `fork()`?
  - this seems to be the correct solution
    - i.e., copy PTEs from parent and *reset* for *copy-on-write* on *all private pages (in all private mapping)*

A Private-Mapped File Changes

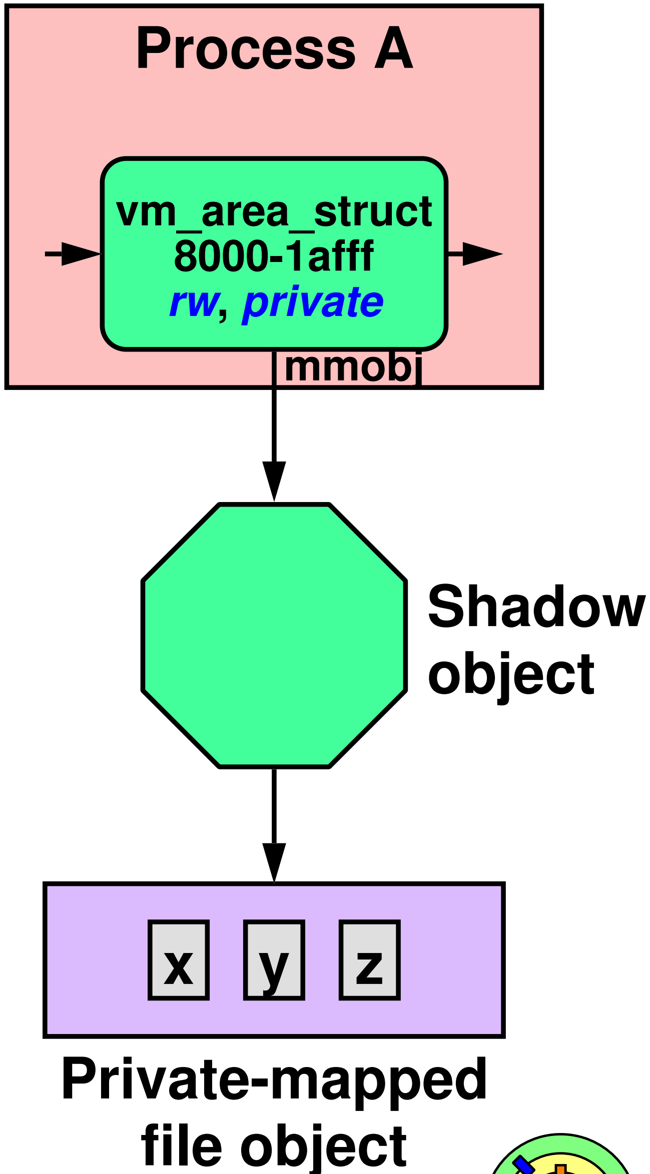


Data = 17;

- Complication: what if the page is modified before `fork()`?
  - but what if the parent or the child calls `fork()` again?
    - afterwards, another process calls `fork()` again, etc.?
    - *cannot* use PTEs to keep track (example later)

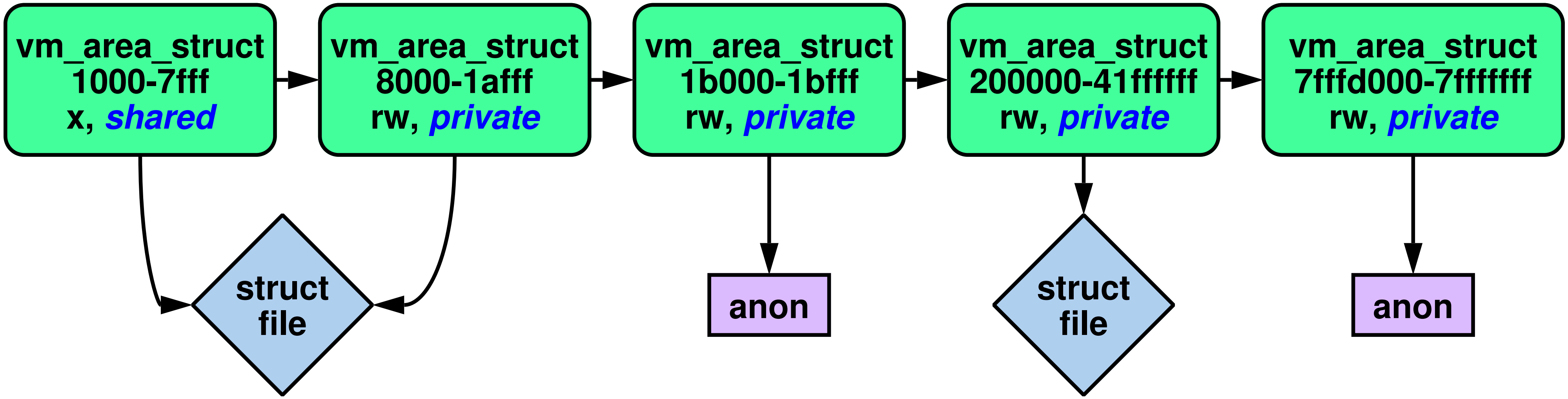
Copy-on-write & Fork

- ***Shadow Objects***
  - indirection
  - keep track of pages that were ***originally copy-on-write*** but have been ***modified***
- **A page in a memory map, into which an object was mapped *private* (e.g., data region), has an associated *shadow object***
  - if a page is "managed by a shadow object" (or "referenced in a shadow object"), it has been modified
  - otherwise, the page is managed by the ***original*** object (file or a "zero/anonymous" object)
  - x, y, z on the right are pages / page frames
- ***Shadow object*** **tells you *where to copy from* when you need to perform *copy-on-write***



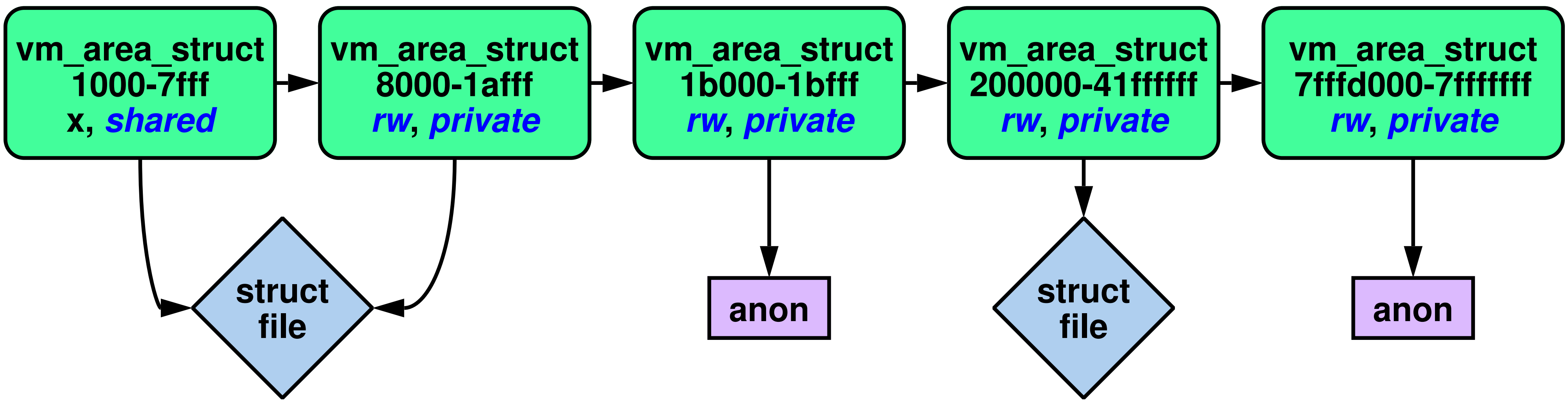
Address-Space Representation

- Remember this?

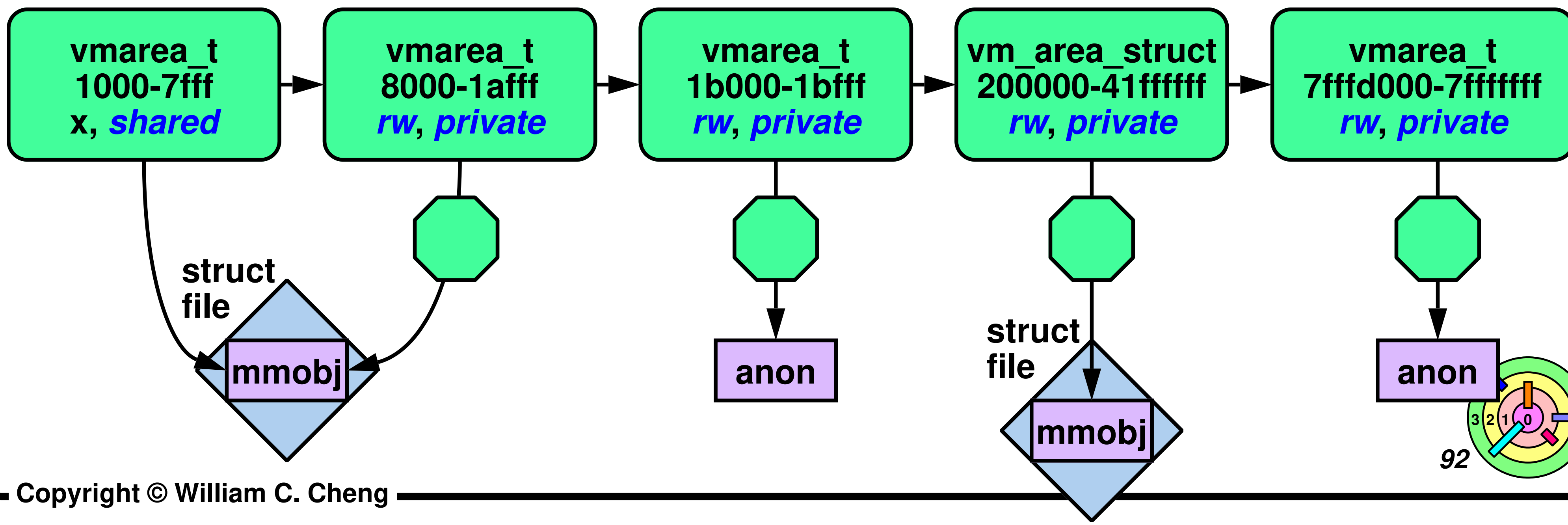


Address-Space Representation

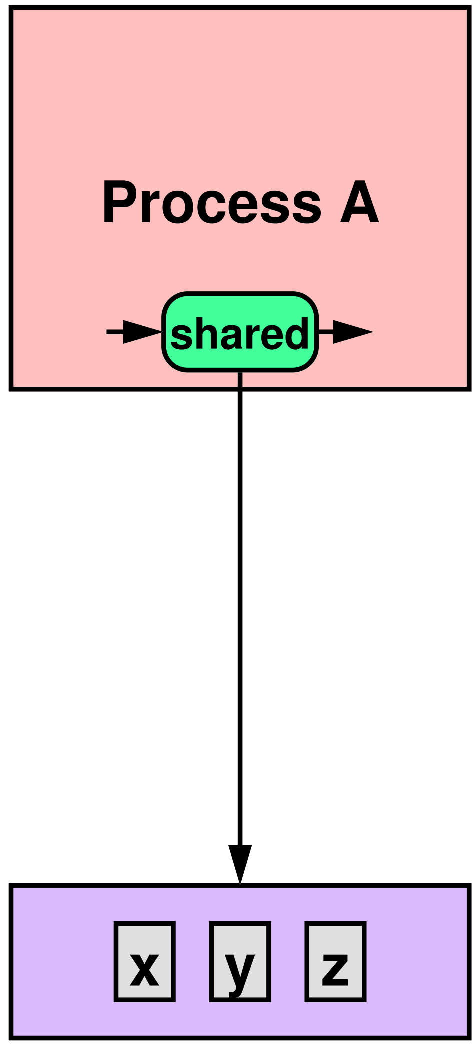
- Remember this?



- now we have to start with (`mmobj` is used in `weenix`):



Share Mapping (1)



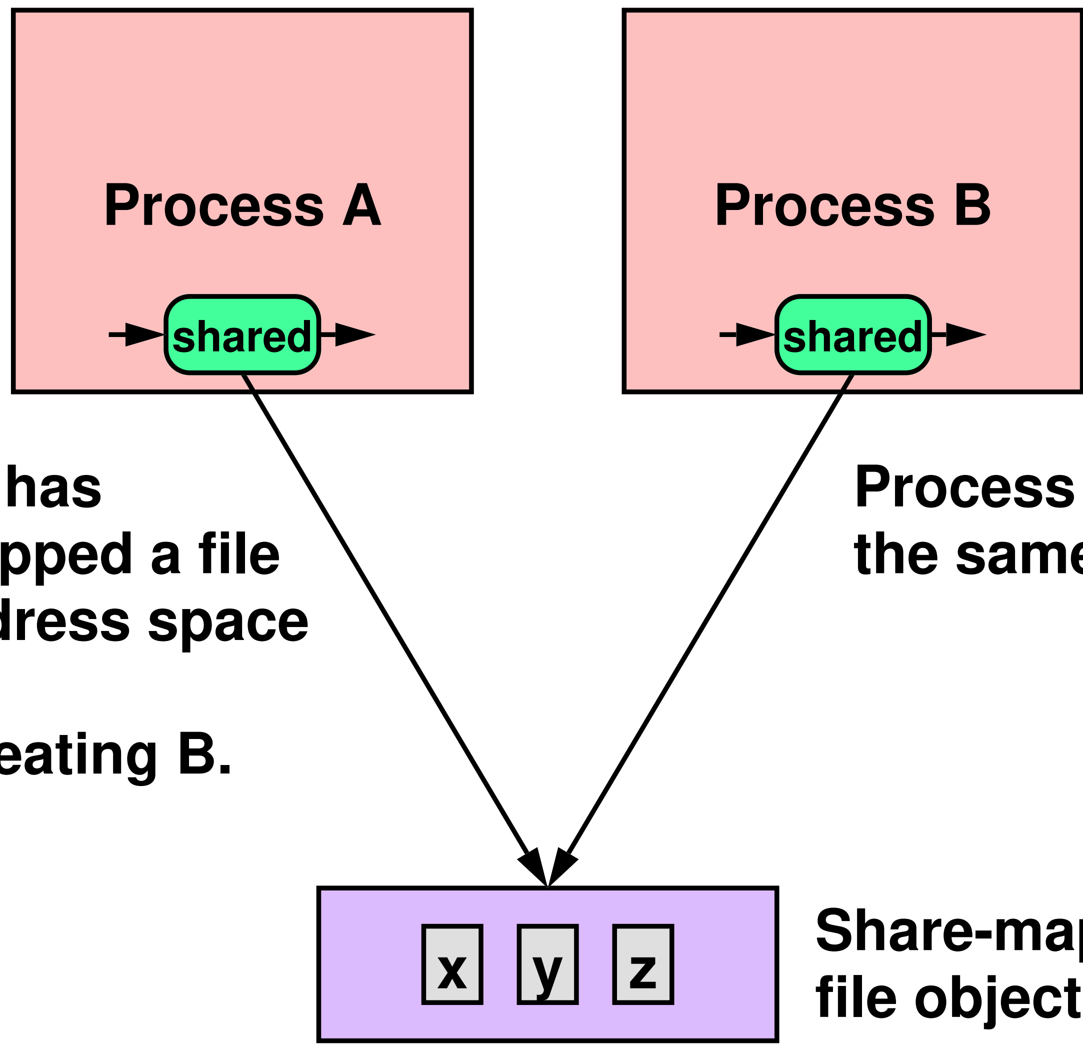
File object

Process A has shared-mapped a file into its address space

in **weenix**

- instead of pointing to a File object, it's pointing to an `mmobj` inside a `vnode` inside a File object
- `mmobj` is used to manage page frames

Share Mapping (2)



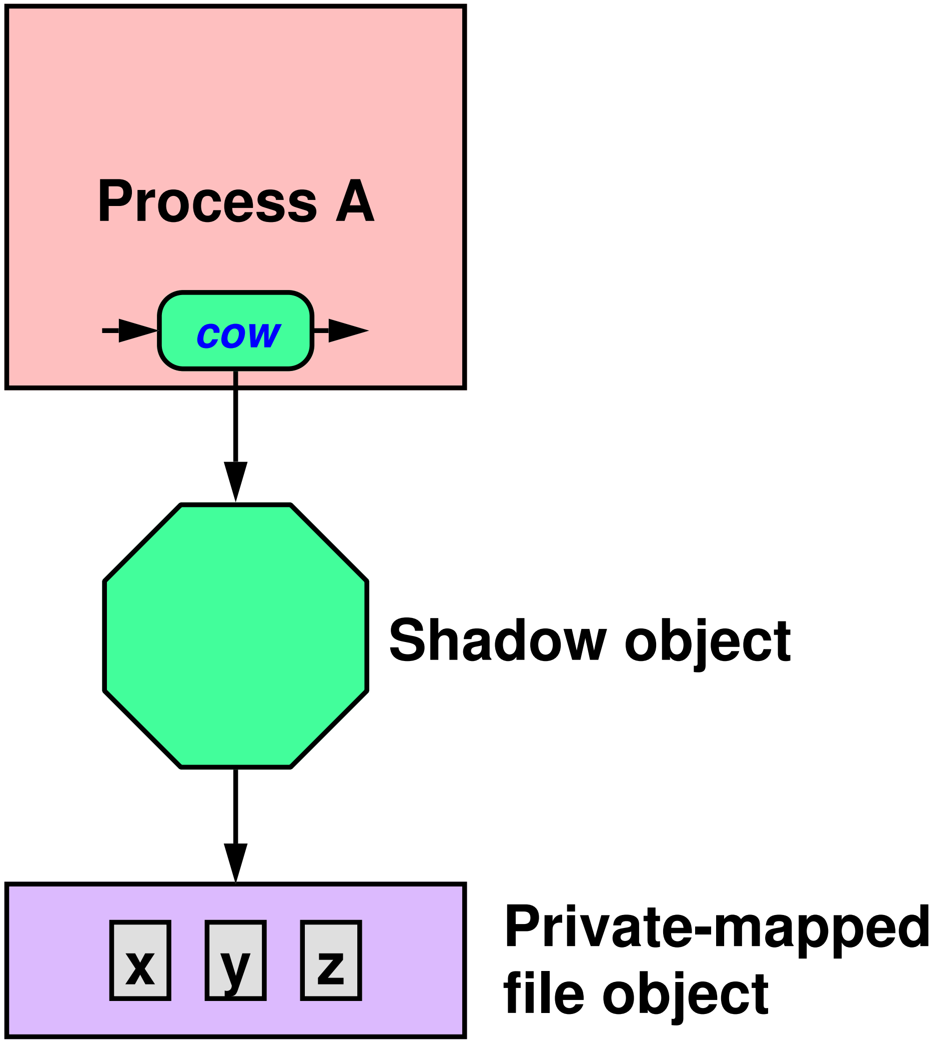
Process A has shared-mapped a file into its address space

A forks, creating B.

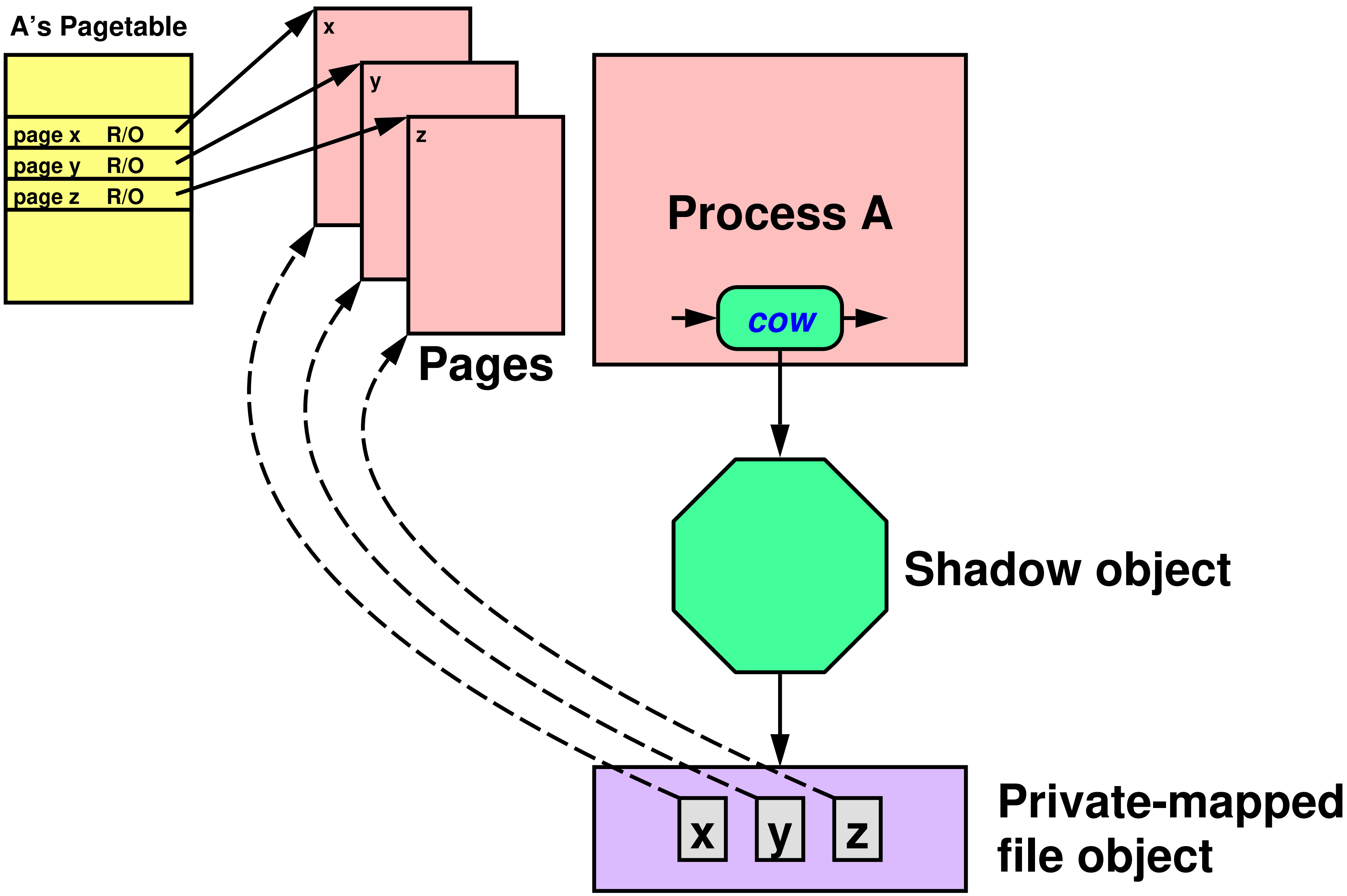
Process B also has the same shared mapping

Share-mapped file object

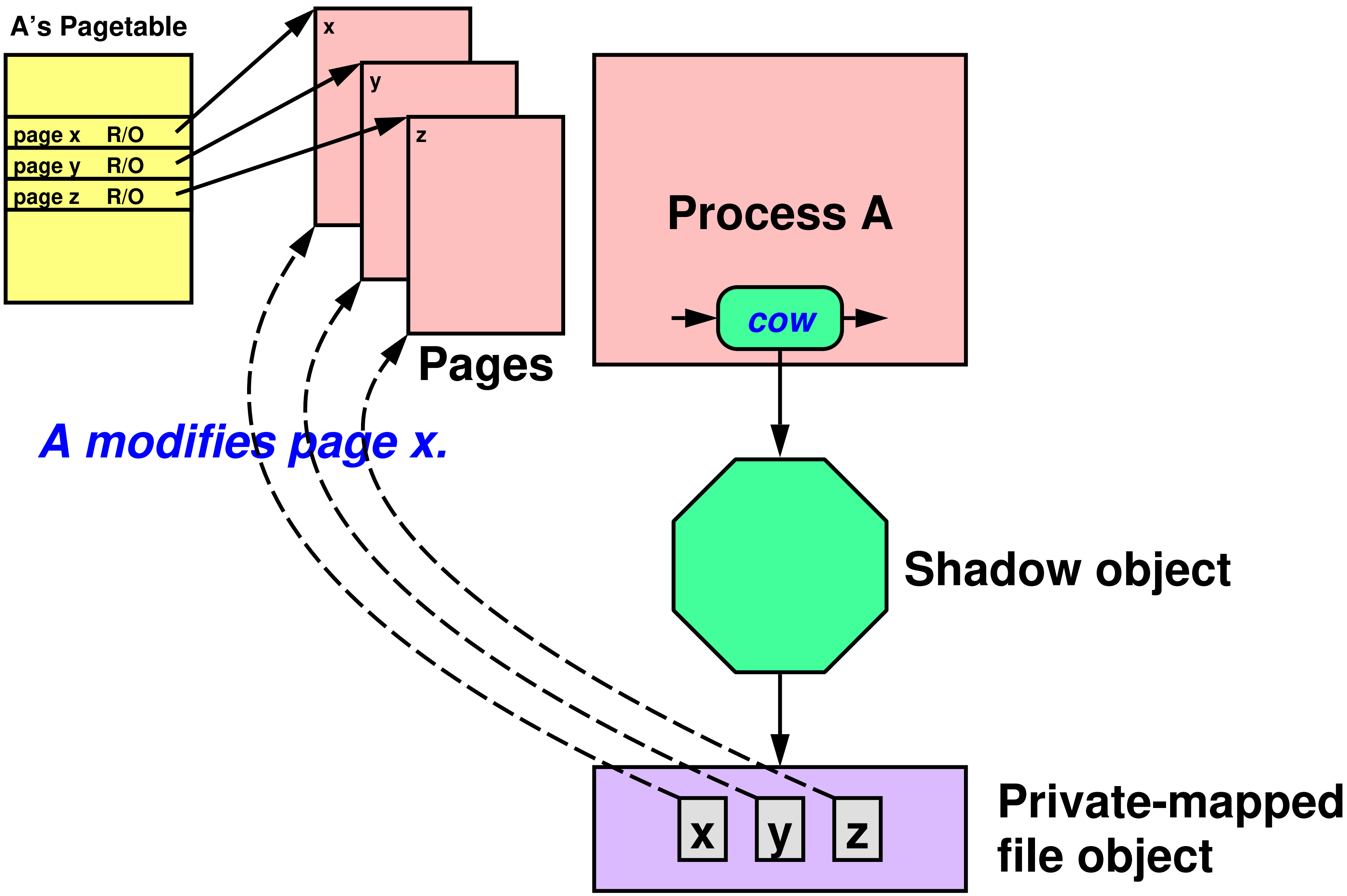
Private Mapping (1)



Private Mapping (1)

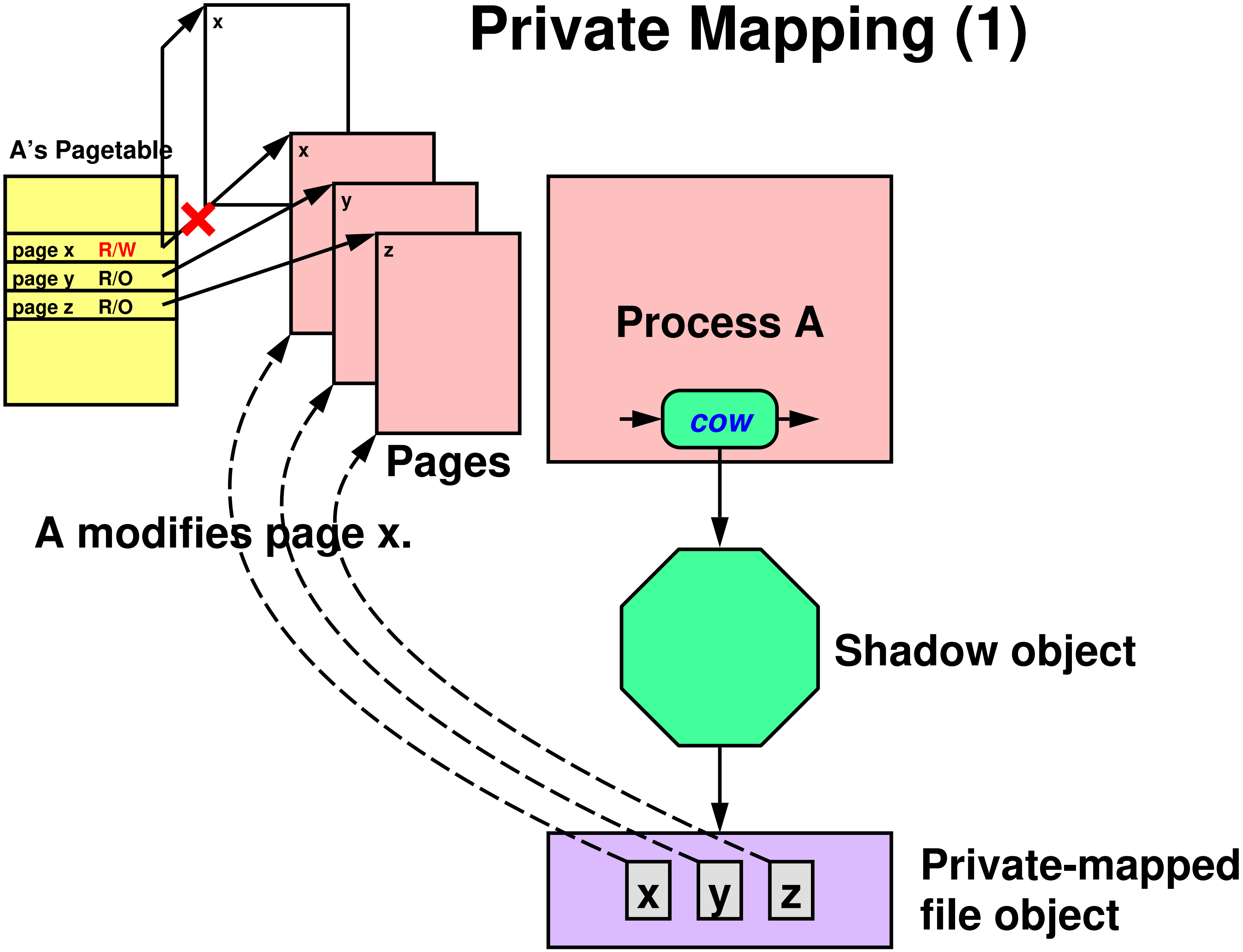


Private Mapping (1)



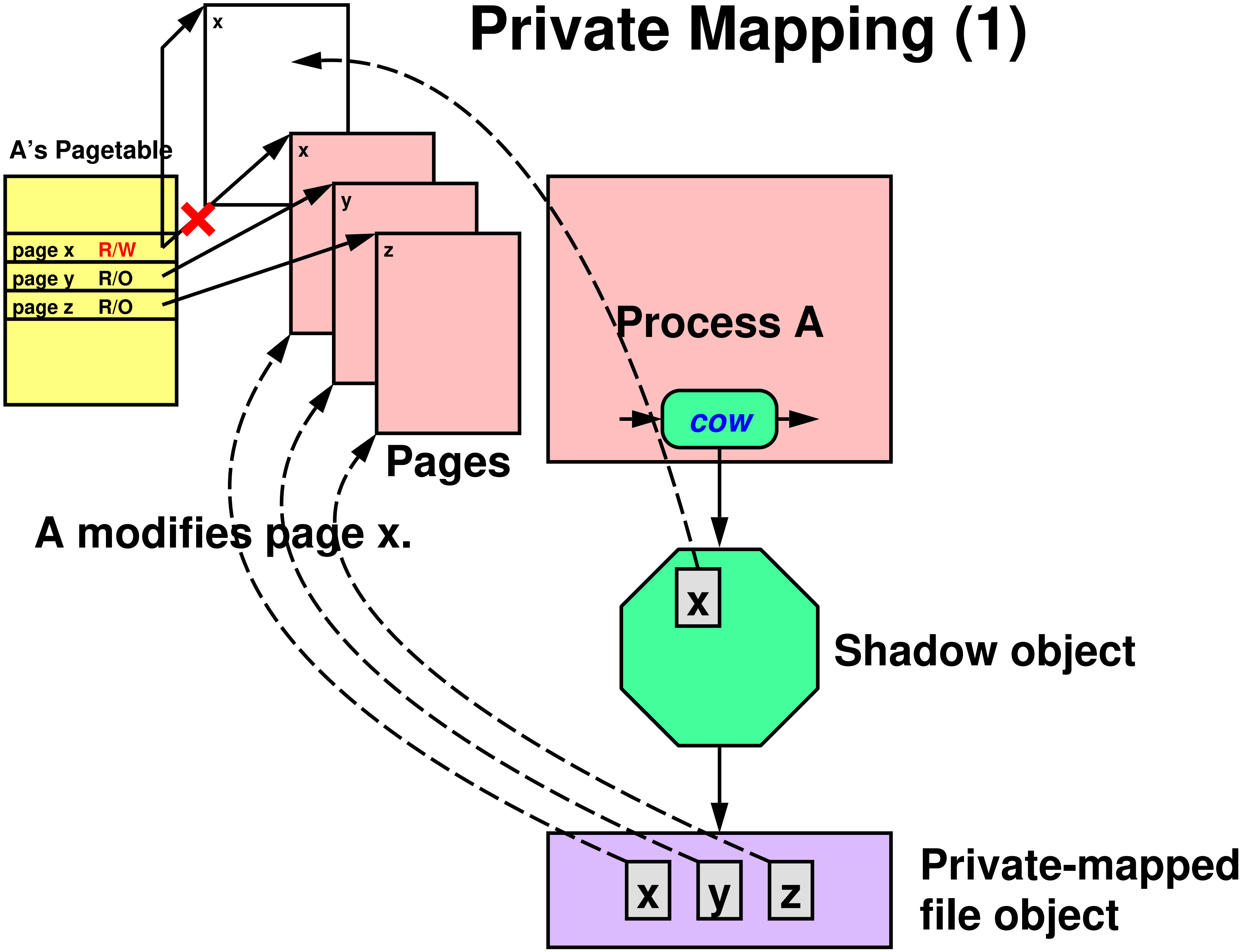
A modifies page x.

Private Mapping (1)



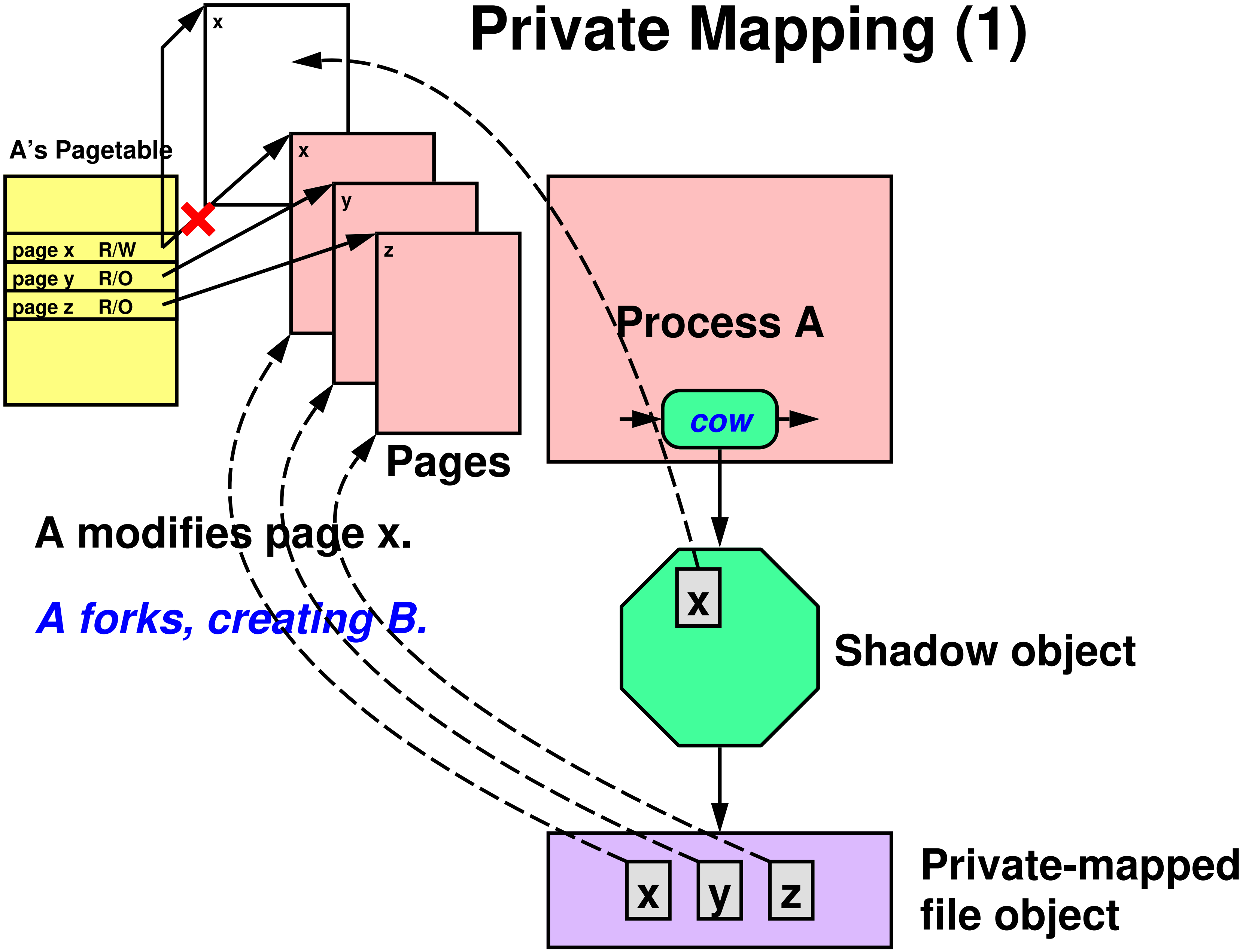
A modifies page x.

Private Mapping (1)



A modifies page x.

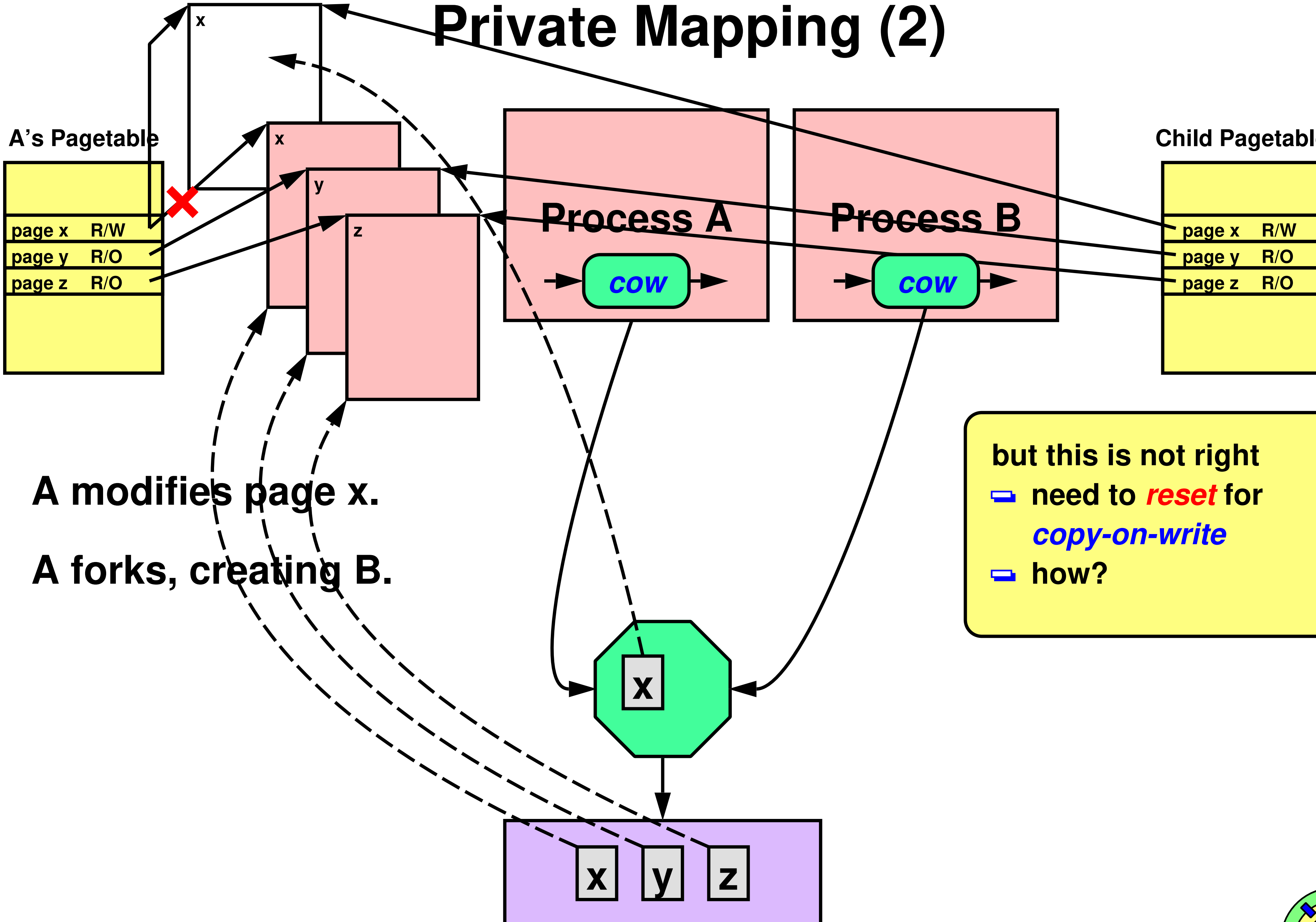
Private Mapping (1)



A modifies page x.

A forks, creating B.

Private Mapping (2)

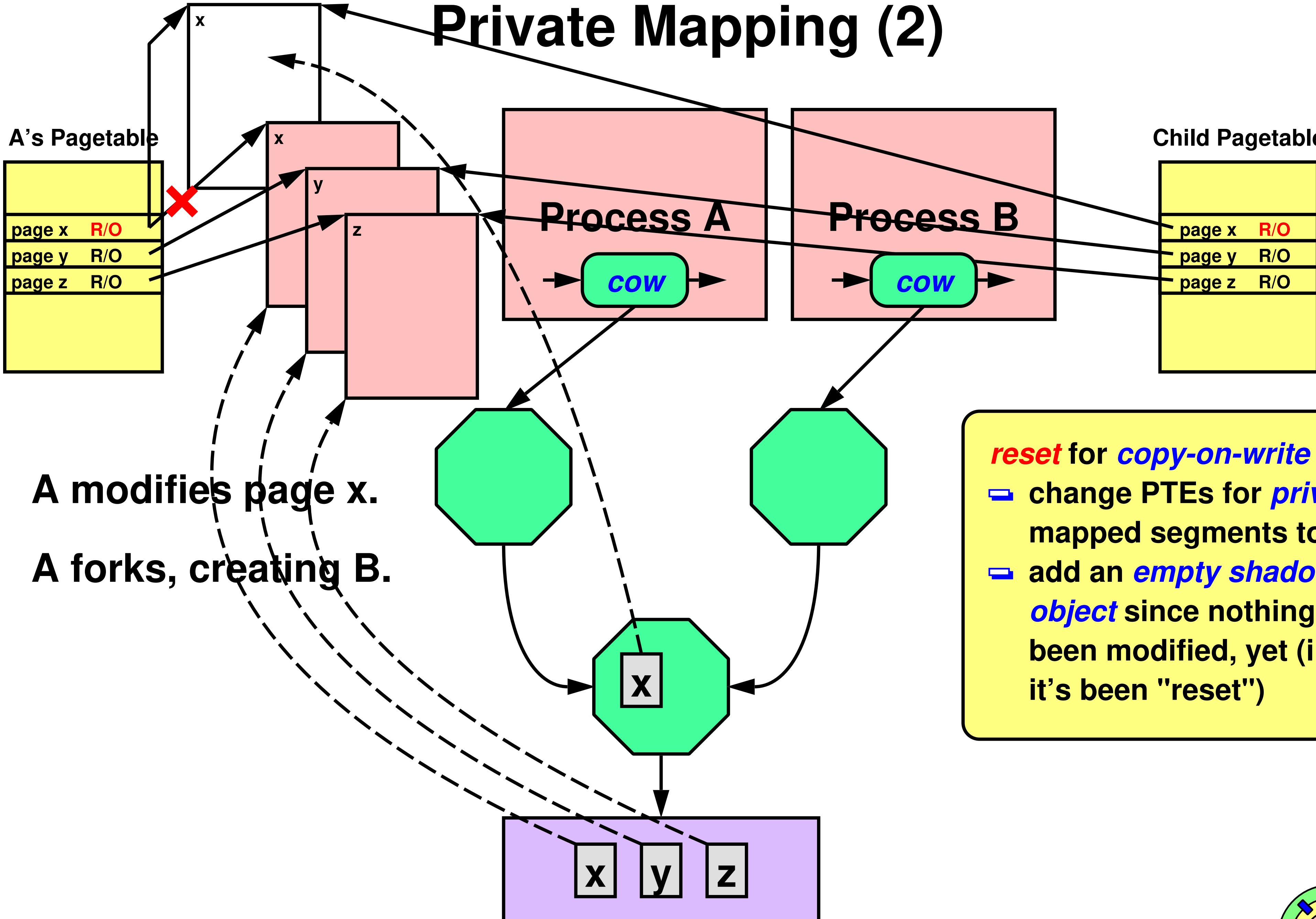


A modifies page x.

A forks, creating B.

but this is not right
- need to *reset* for *copy-on-write*
- how?

Private Mapping (2)



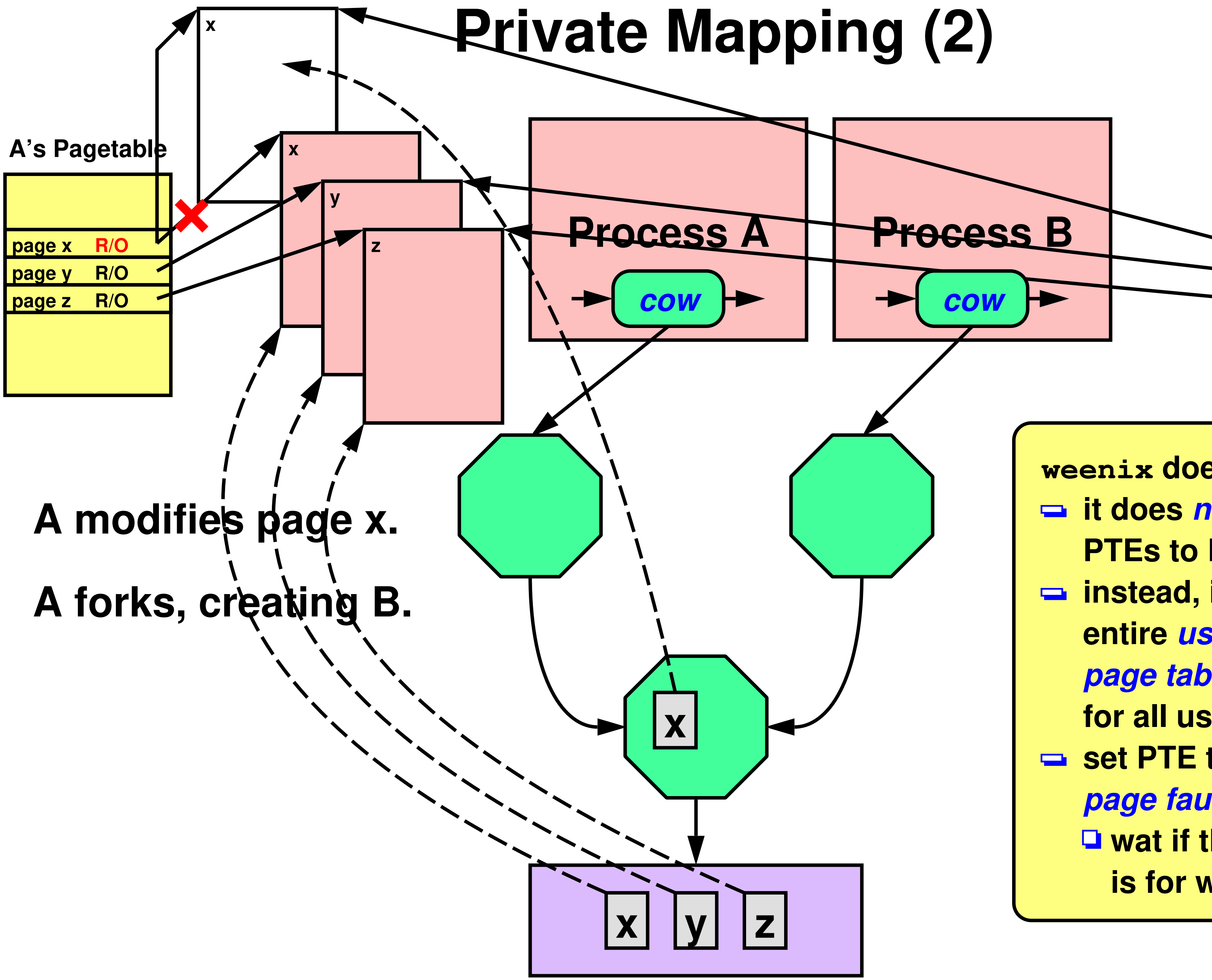
A modifies page x.

A forks, creating B.

reset **for** ***copy-on-write***

- **change PTEs for** ***privately*** **mapped segments to** ***R/O***
- **add an** ***empty shadow object*** **since nothing has been modified, yet (i.e., it's been "reset")**

Private Mapping (2)



A modifies page x.

A forks, creating B.

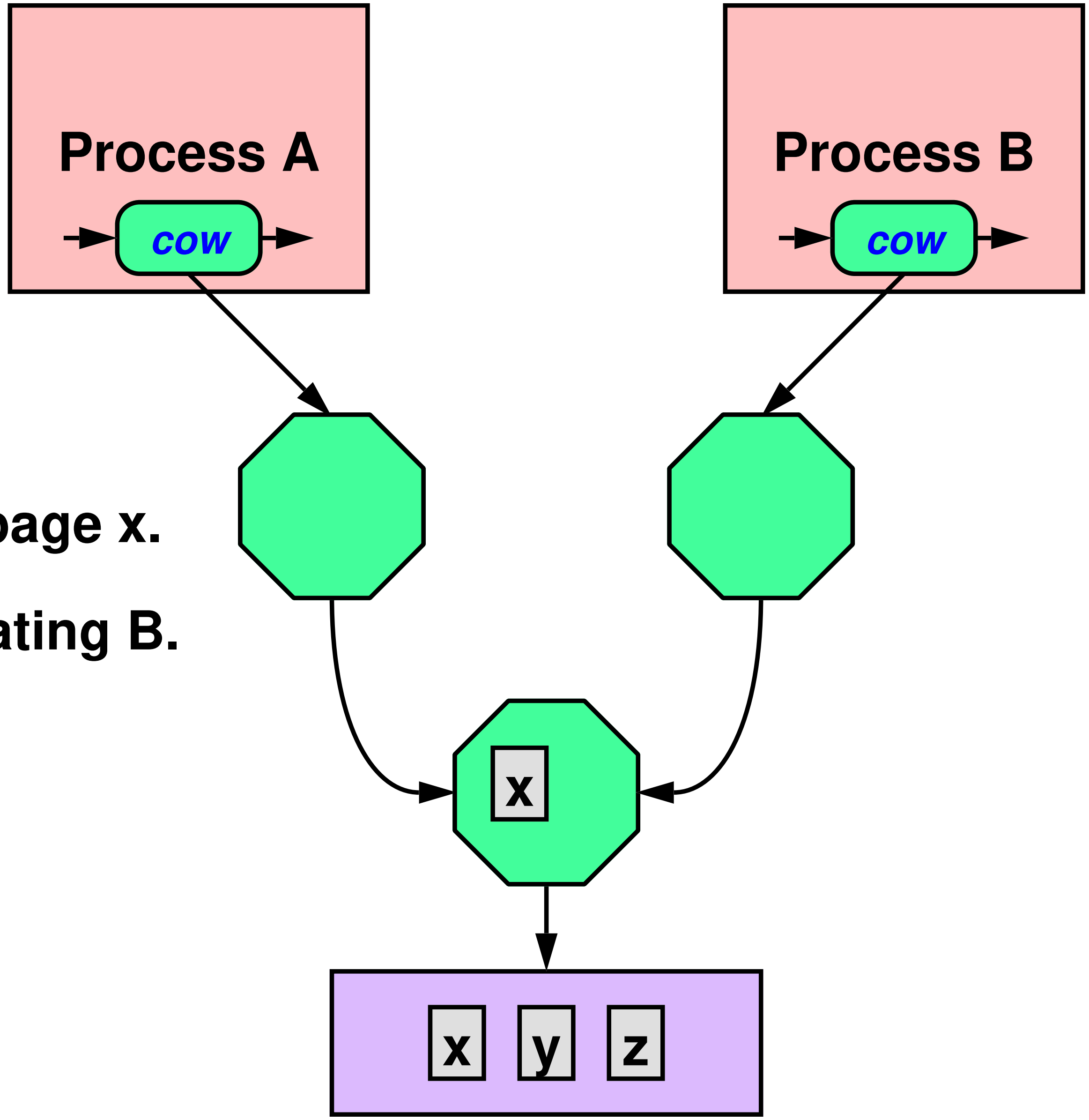
`weenix` does it in *two steps*

- it does *not* change these PTEs to R/O
- instead, it *unmaps* the entire *user space page table* (i.e., sets V=0 for all user space PTEs)
- set PTE to R/O on the *next page fault* if reading
  - wat if the next page fault is for writing?

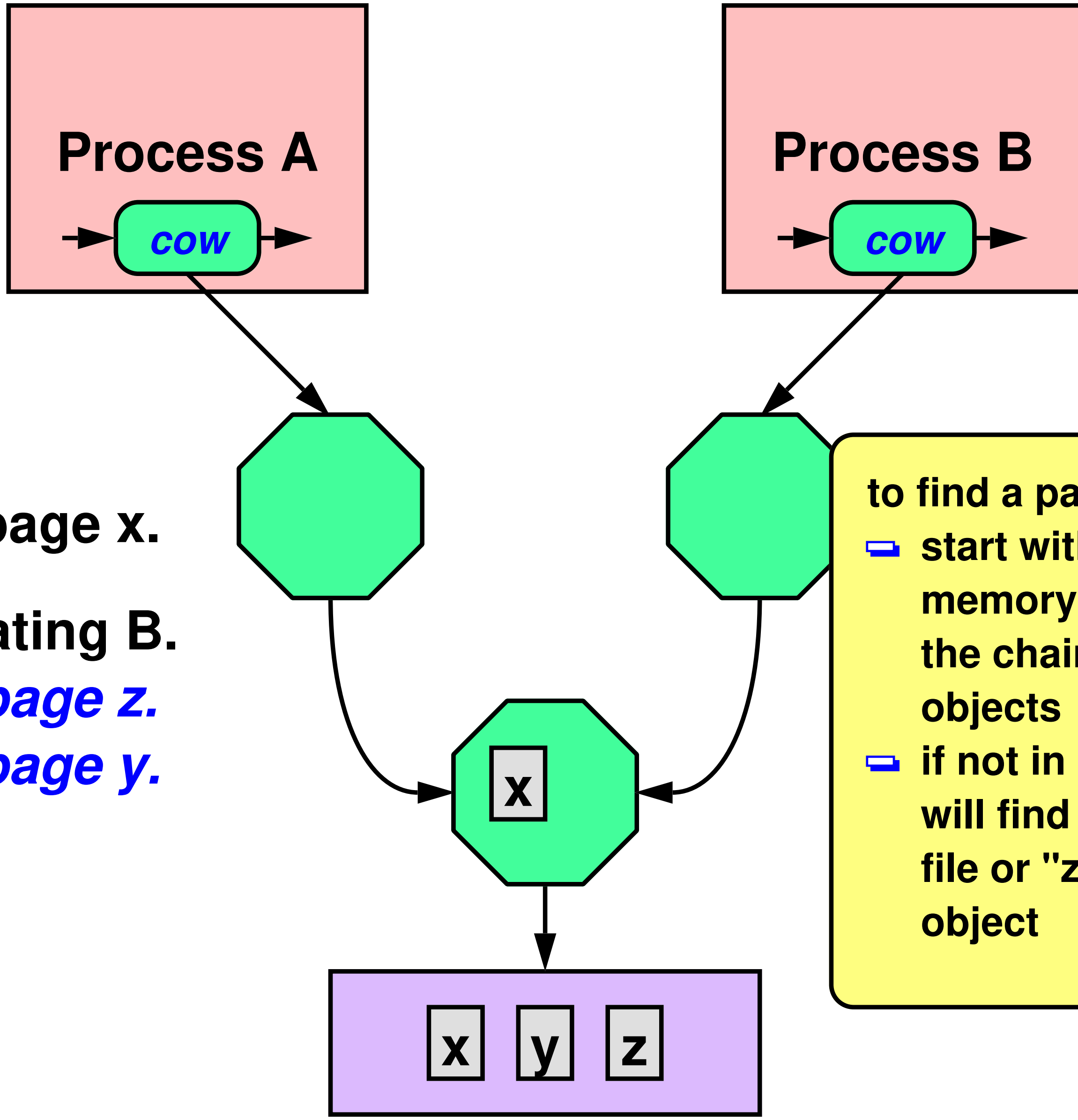
Private Mapping (2)

A modifies page x.

A forks, creating B.



Private Mapping (2)



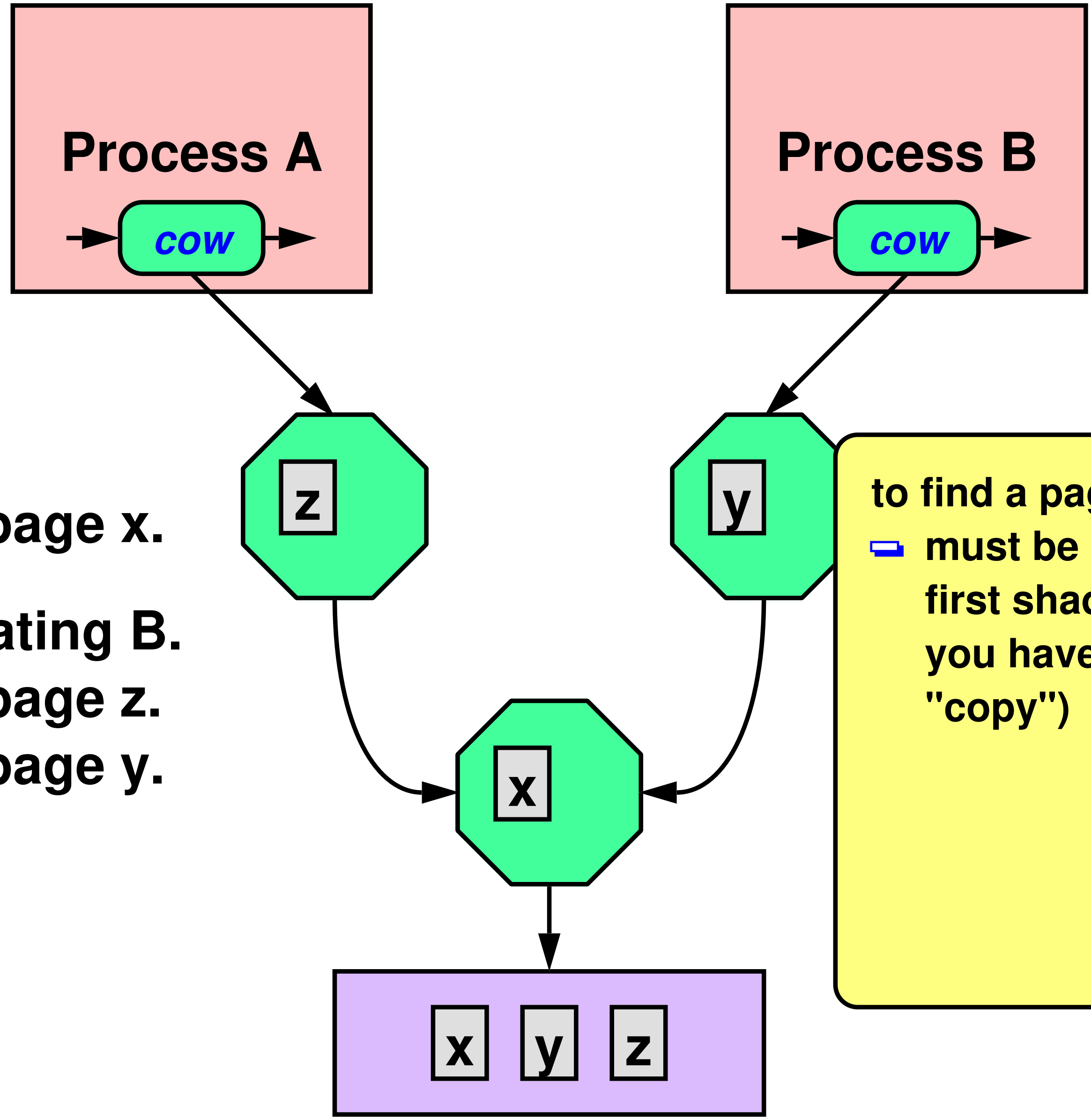
A modifies page x.

A forks, creating B.
A modifies page z.
B modifies page y.

to find a page to *copy from*

- start with the process' memory map and follow the chain of shadow objects
- if not in a shadow object, will find it in the mapped file or "zero/anonymous" object

Private Mapping (2)

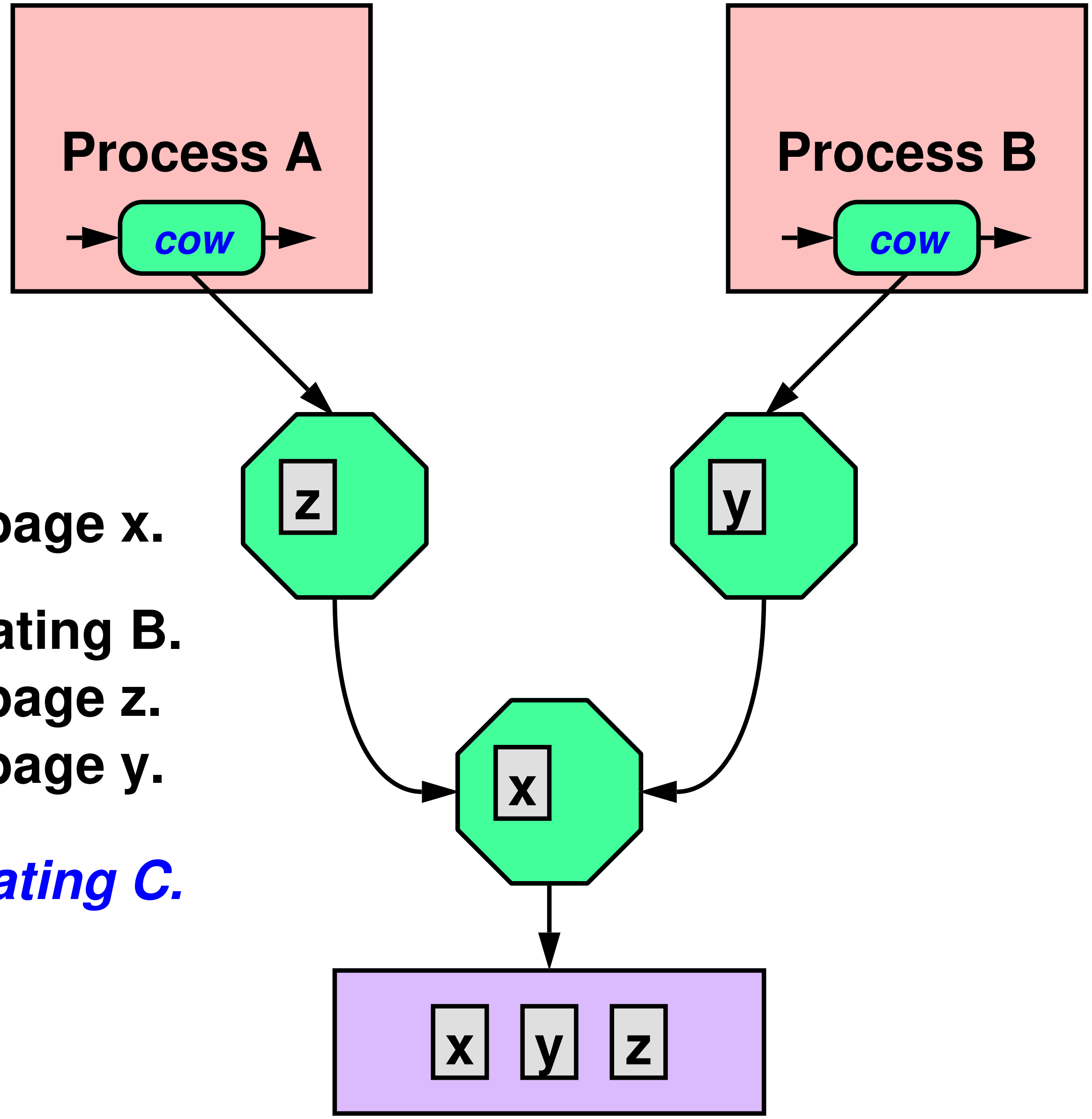


A modifies page x.

A forks, creating B.
A modifies page z.
B modifies page y.

to find a page to *copy to*
- must be managed by the first shadow object (after you have performed "copy")

Private Mapping (2)

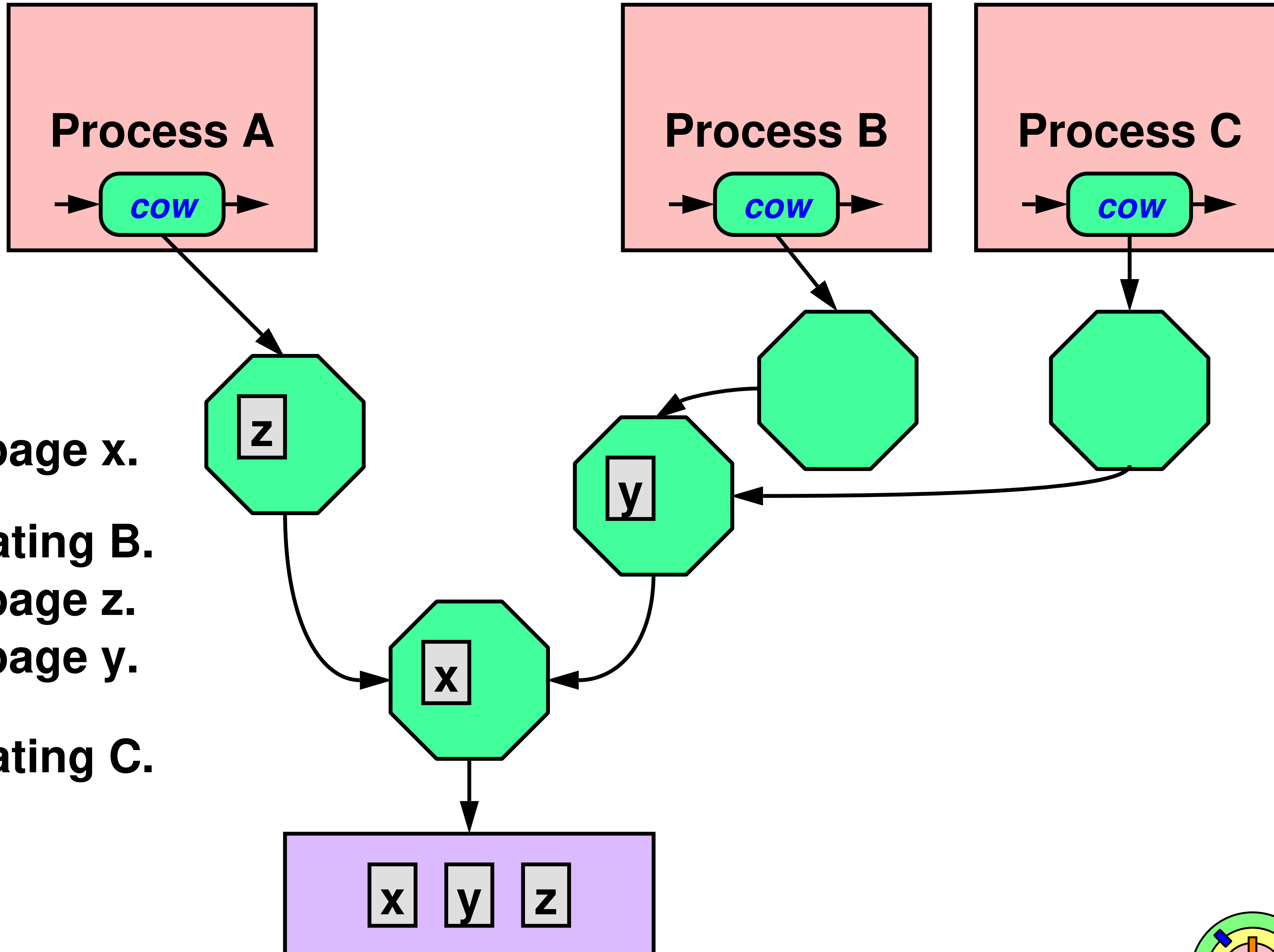


A modifies page x.

A forks, creating B.
A modifies page z.
B modifies page y.

B forks, creating C.

Private Mapping (3)

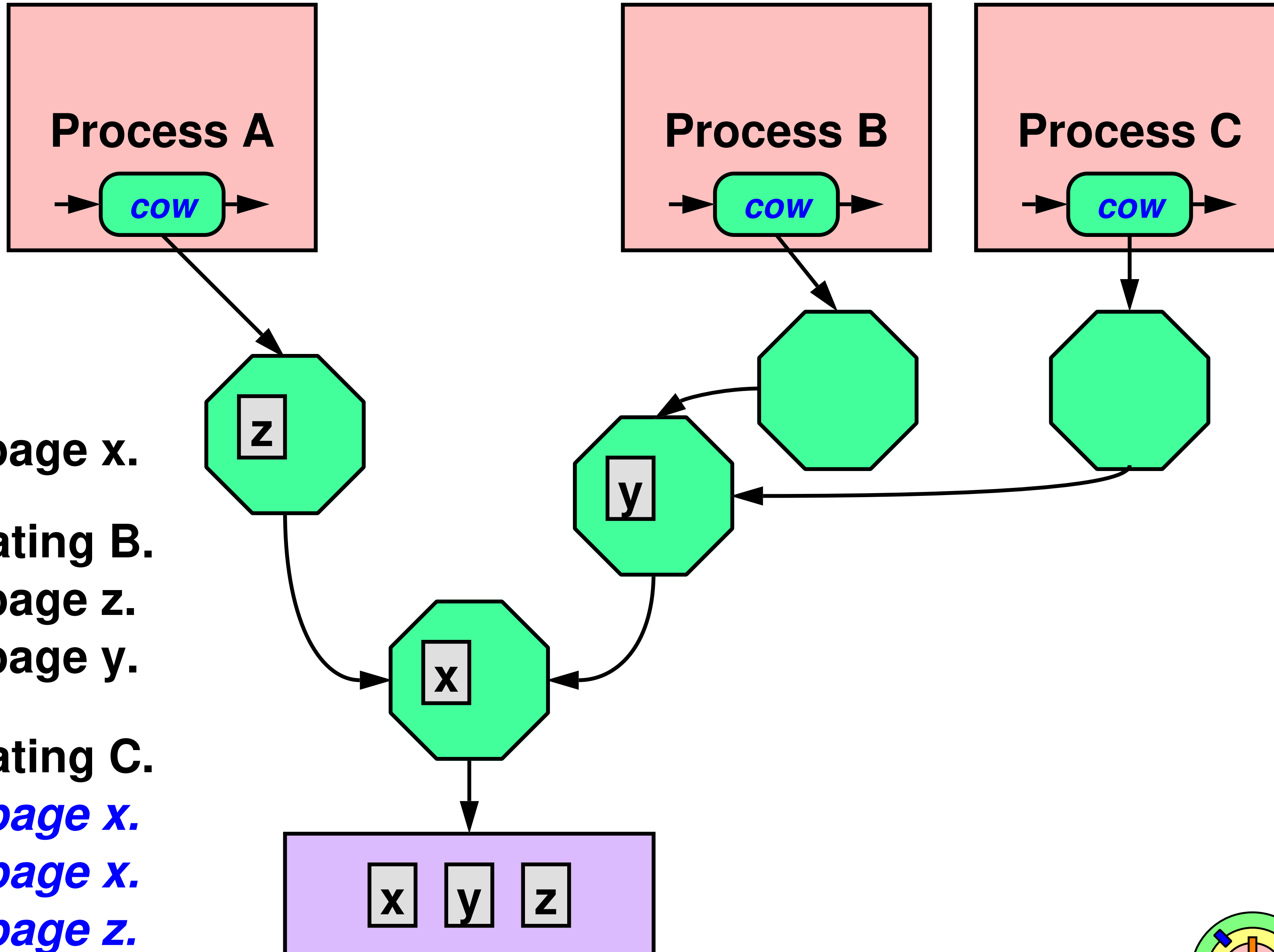


A modifies page x.

A forks, creating B.
A modifies page z.
B modifies page y.

B forks, creating C.

Private Mapping (3)

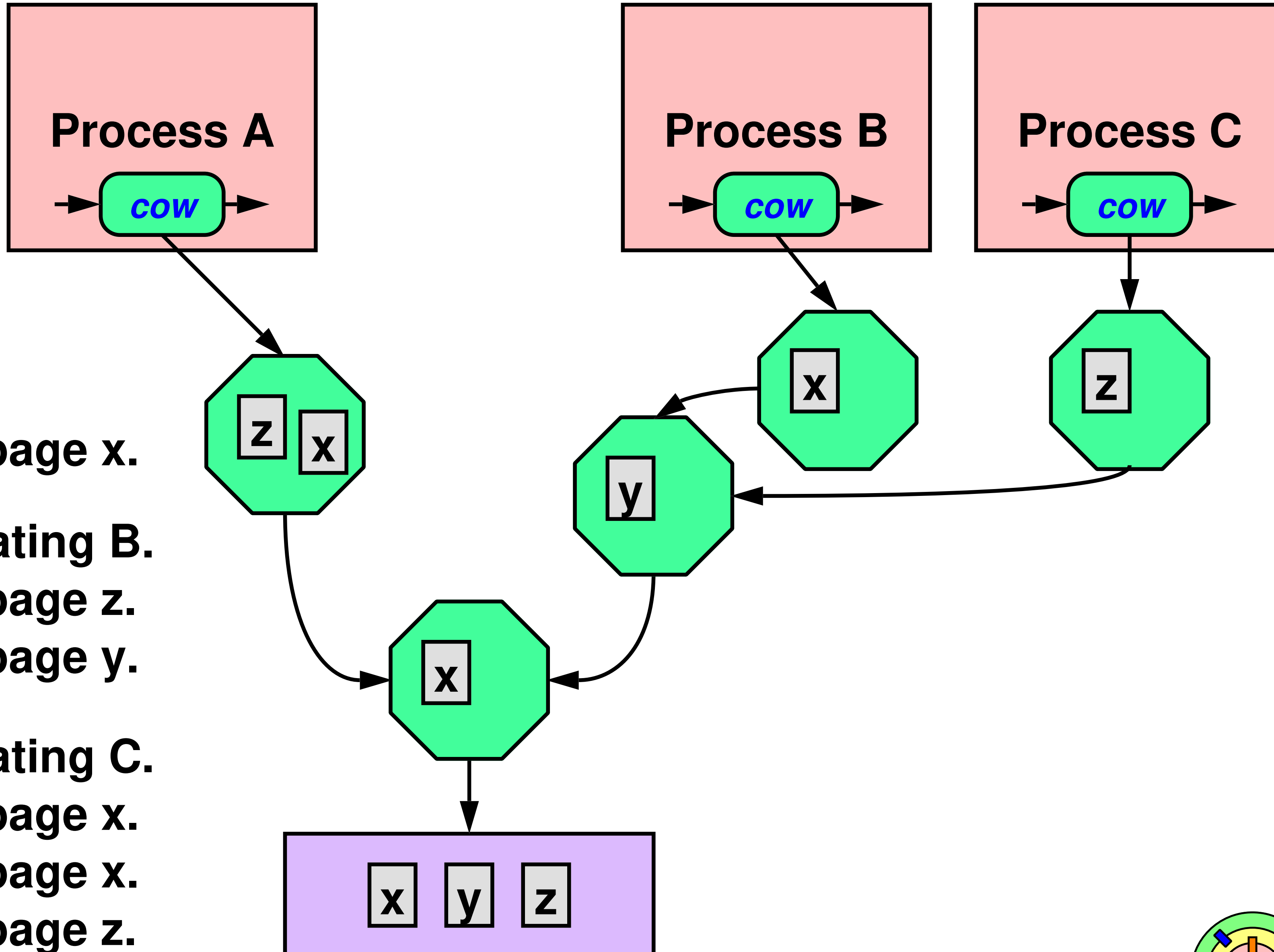


A modifies page x.

A forks, creating B.
A modifies page z.
B modifies page y.

B forks, creating C.
A modifies page x.
B modifies page x.
C modifies page z.

Private Mapping (3)

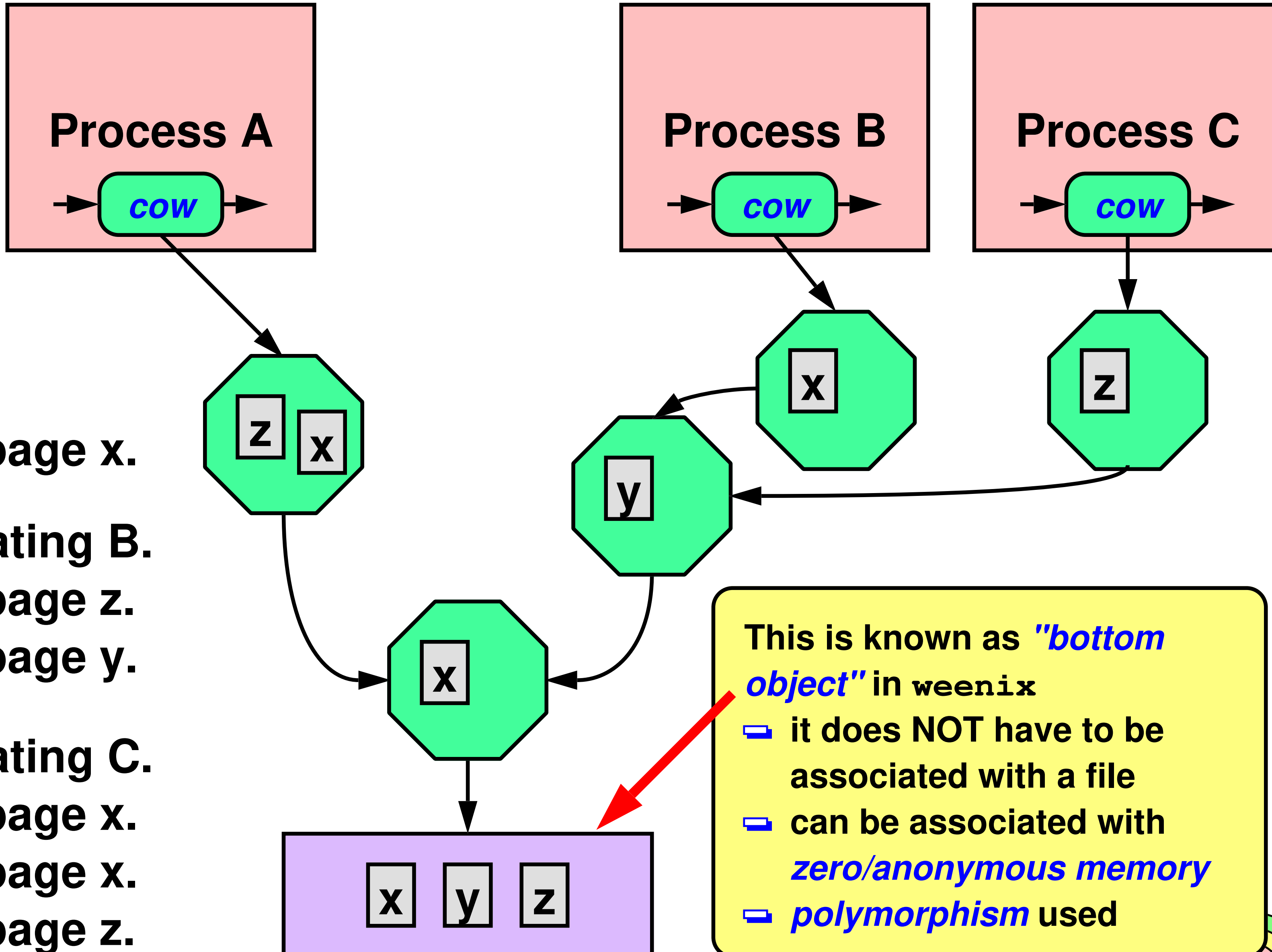


A modifies page x.

A forks, creating B.
A modifies page z.
B modifies page y.

B forks, creating C.
A modifies page x.
B modifies page x.
C modifies page z.

Private Mapping (3)



A modifies page x.

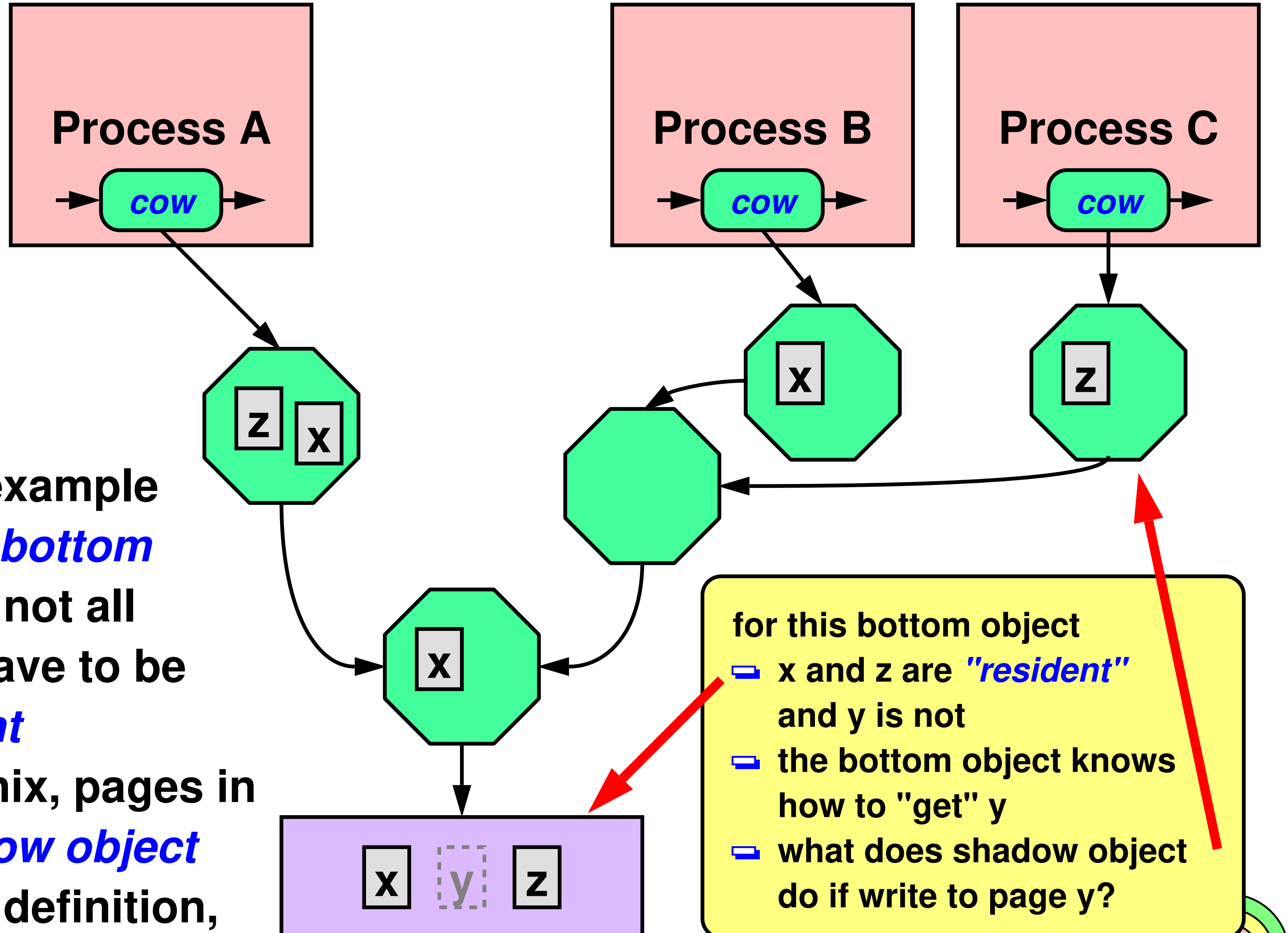
A forks, creating B.
A modifies page z.
B modifies page y.

B forks, creating C.
A modifies page x.
B modifies page x.
C modifies page z.

This is known as *"bottom object"* in `weenix`

- it does NOT have to be associated with a file
- can be associated with *zero/anonymous memory*
- *polymorphism* used

Private Mapping (4)



- a slightly different example
  - for the ***bottom object***, not all page have to be ***resident***
  - in weenix, pages in a ***shadow object*** are, by definition, ***"resident"***

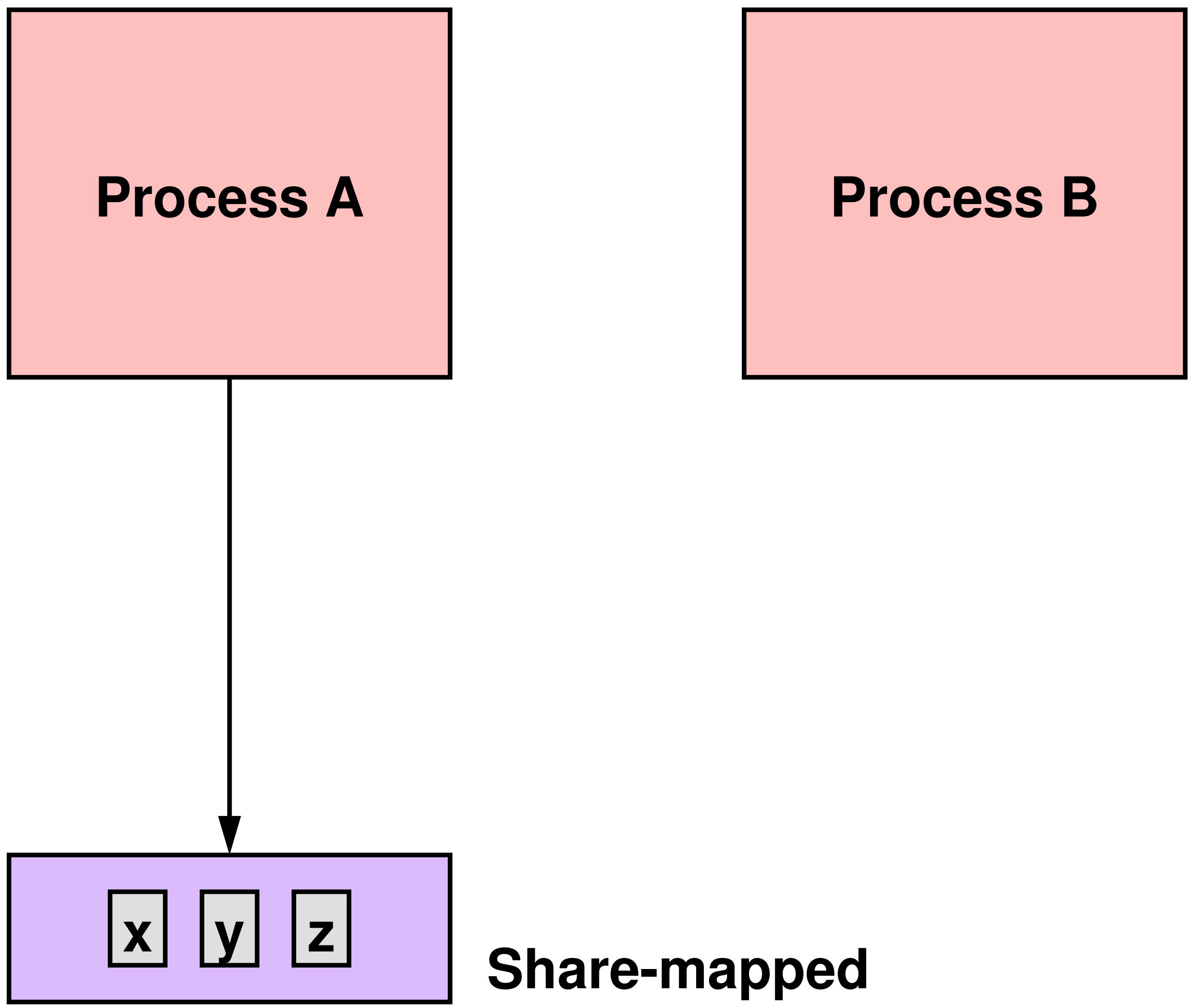
for this bottom object

- x and z are ***"resident"*** and y is not
- the bottom object knows how to "get" y
- what does shadow object do if write to page y?

Virtual Copy

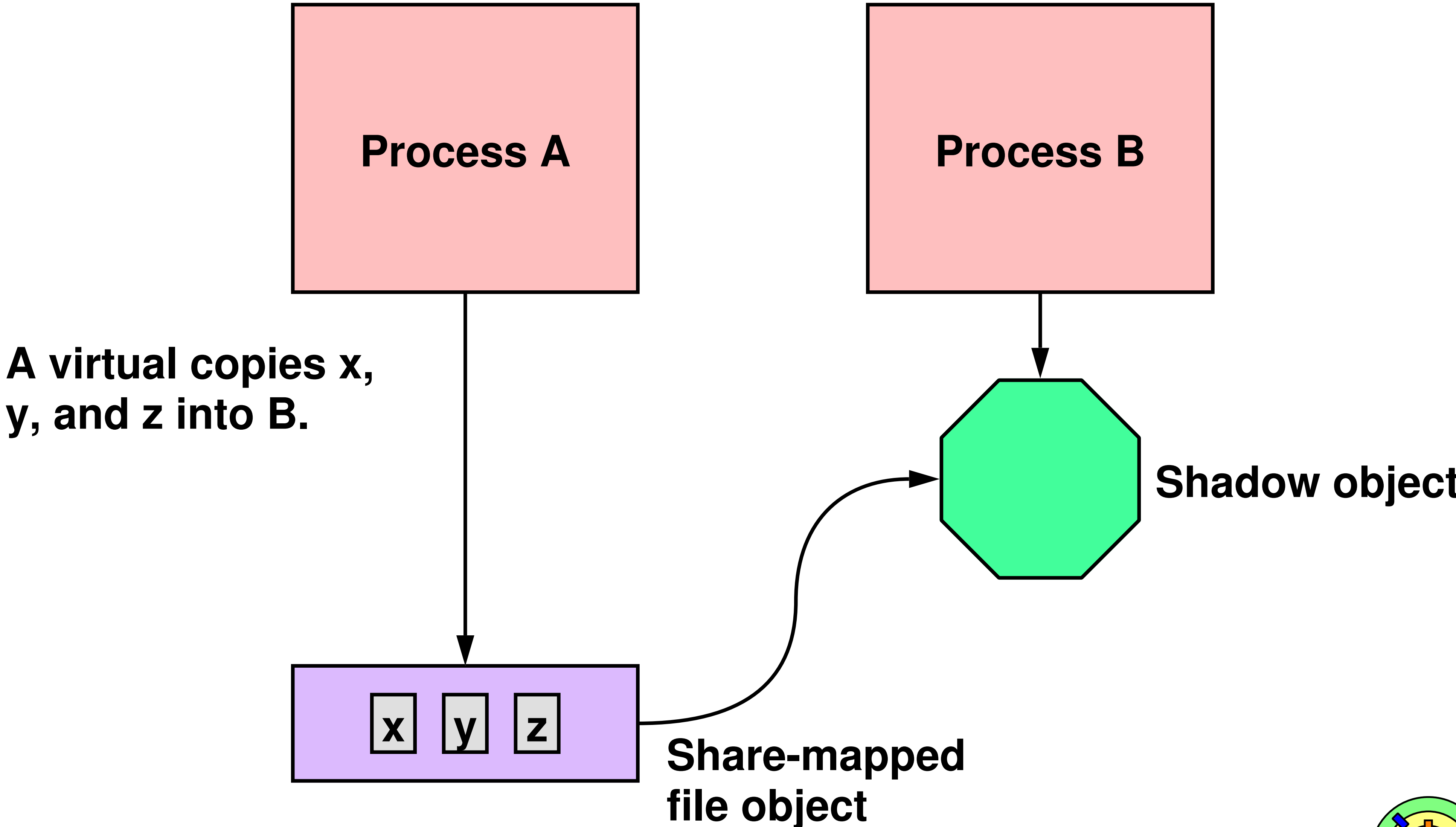
- Local RPC
  - "copy" arguments from one process to another
  - assume arguments are page-aligned and page-sized
  - map pages into both caller and callee, copy-on-write
    - works in most cases, except when the page corresponds to a shared memory-mapped file
      - in this case, the sender does not have a shadow object!

Share and Private Mapping



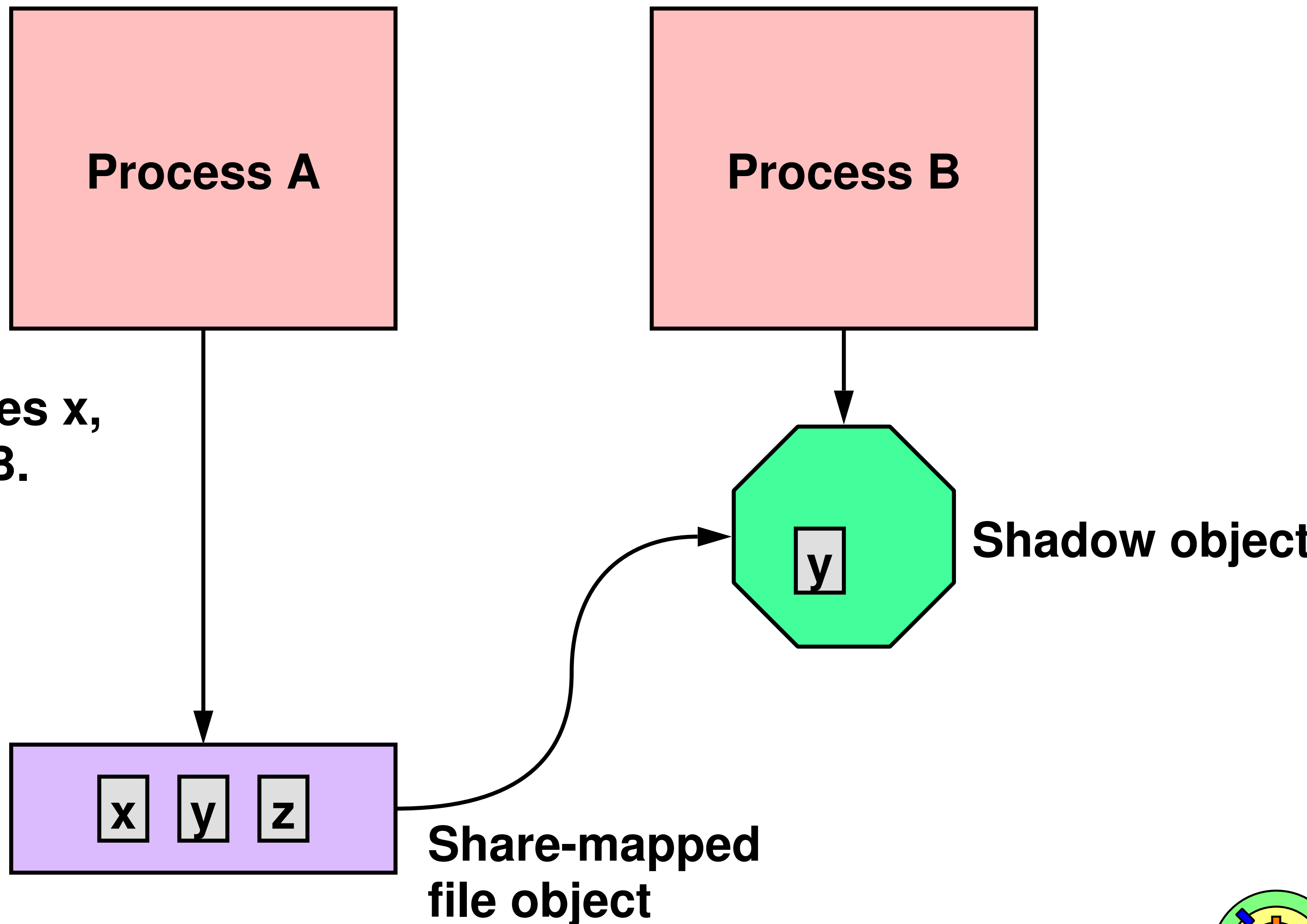
Share-mapped file object

Share and Private Mapping

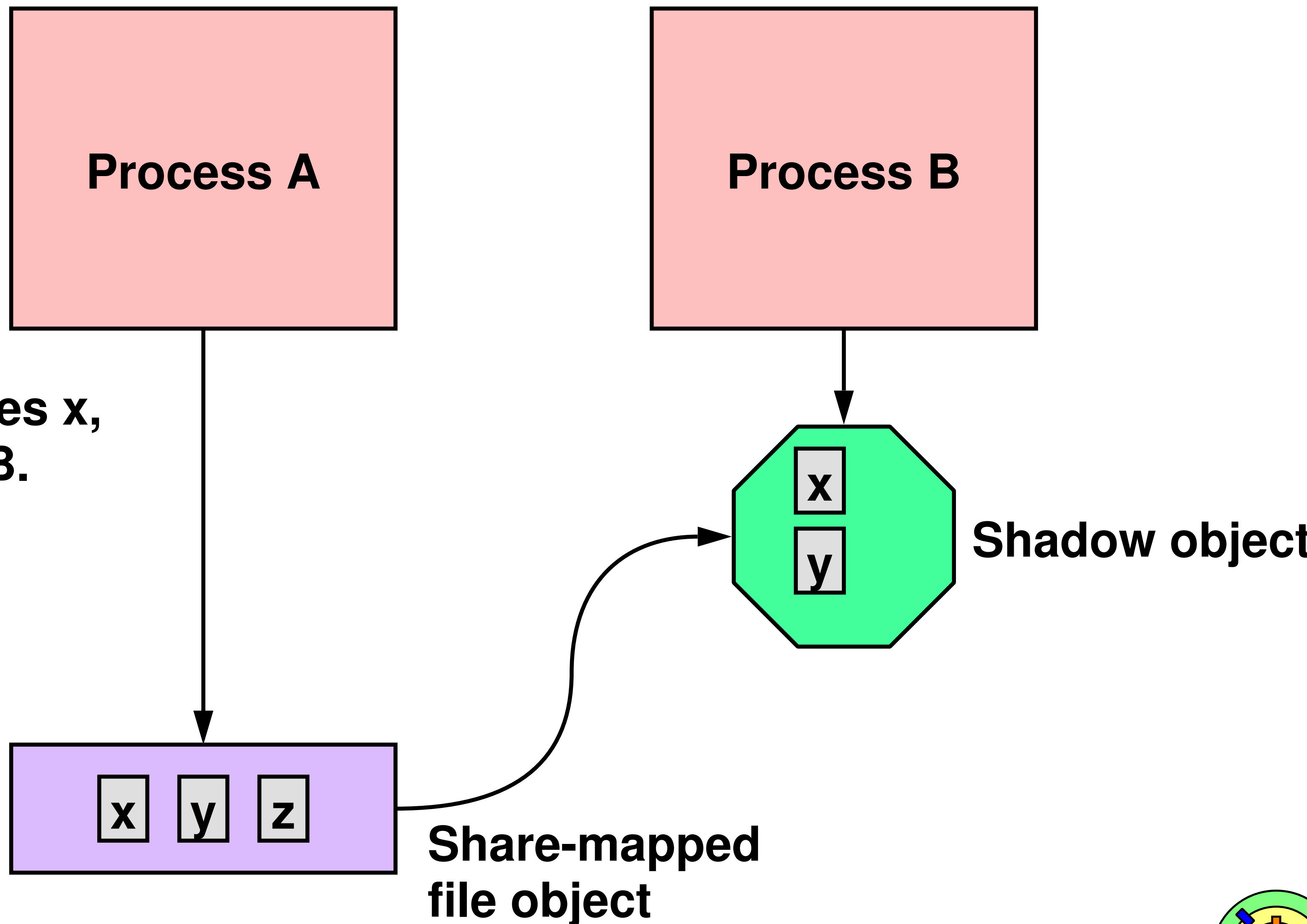


Share and Private Mapping

A virtual copies x, y, and z into B.
B modifies y.

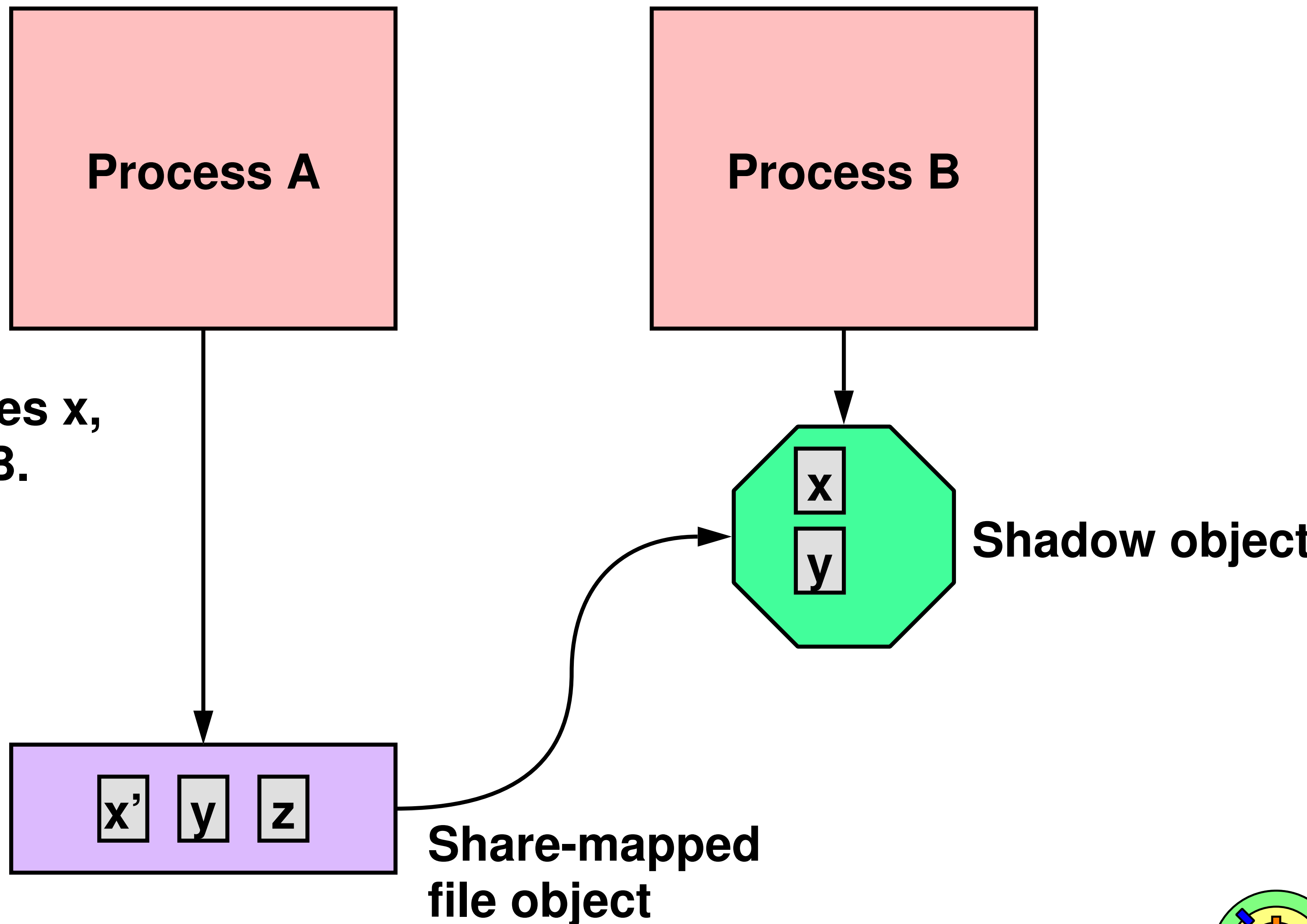


Share and Private Mapping



A virtual copies x, y, and z into B.
B modifies y.
A modifies x.

Share and Private Mapping



A virtual copies x, y, and z into B.
B modifies y.
A modifies x.

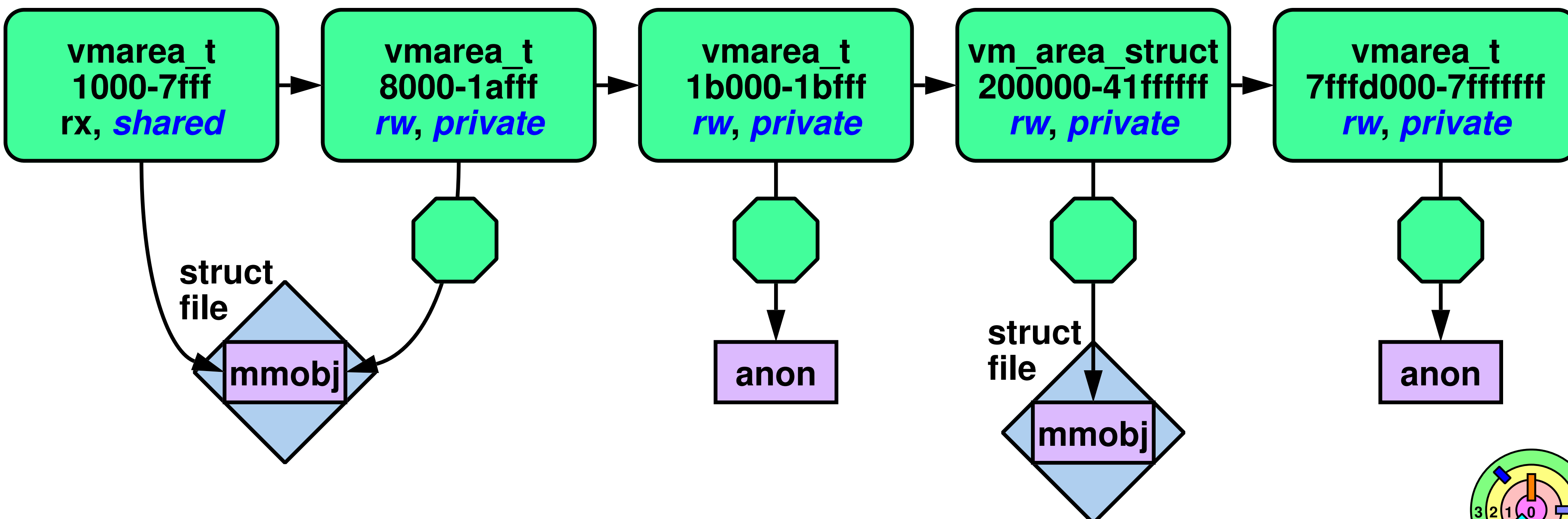
Shadow Objects Summary

- Why go through all this trouble?
  - because we want to implement *copy-on-write* together with `fork()`
    - *a variable* (such as `Data` a few slides back) can exist in *many* different physical pages *simultaneously*
      - each contains a different *version* of this variable

- To manage this mess, `weenix` uses the idea of Shadow Objects
  - what is the "idea" of Shadow Objects?
    - organize a tree of shadow objects using an *inverted tree* data structure
      - where the root is the *bottom object*
    - the rule for finding page frame / physical page that contains the global variable in question for a particular process
      - traversing shadow object pointers on the inverted tree
    - when and how to perform *copy-on-write*
  - you have to implement what's described on these slides in kernel 3

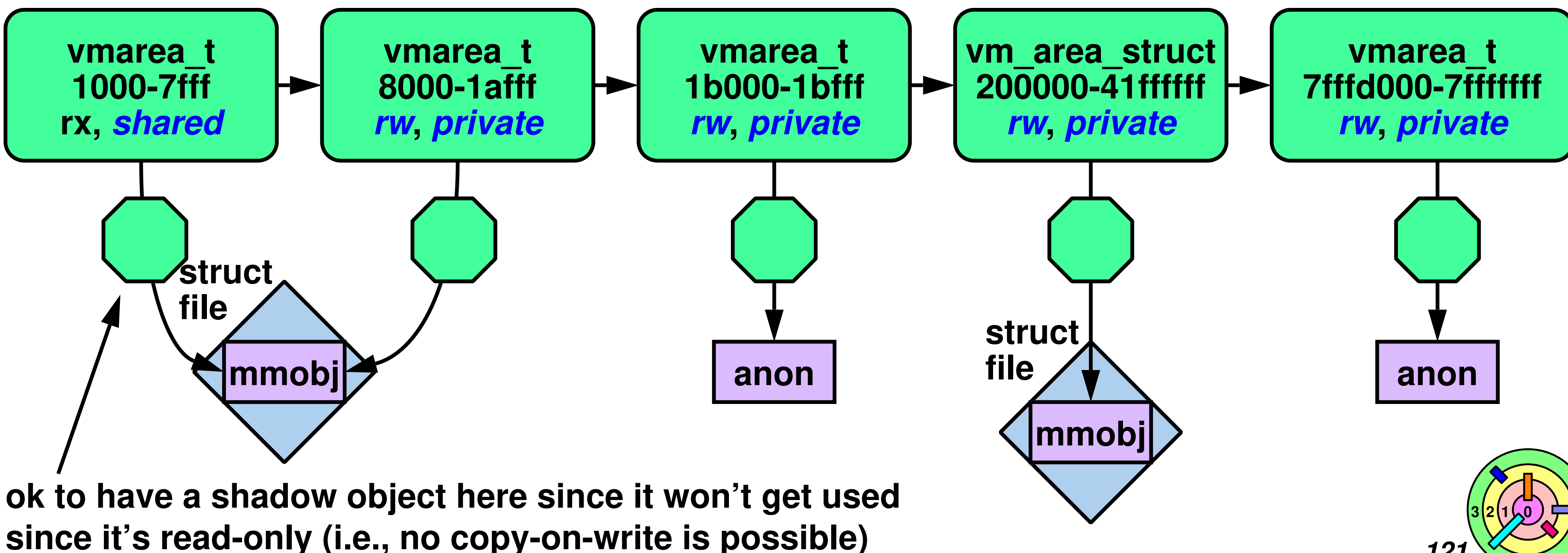
Memory Management Objects in `weenix`

- In `weenix`, an *mmobj* is used to manage *page frames*
  - types of *mmobj* in kernel assignments are:
    - there's one that lives *inside a vnode* (`vn->vn_mmobj`)
    - a *shadow object* is an mmobj
    - an *anonymous object* (meaning not associated with a file and not a shadow object) is an mmobj
  - a `vmarea` is supported by one of these 3 mmobjs



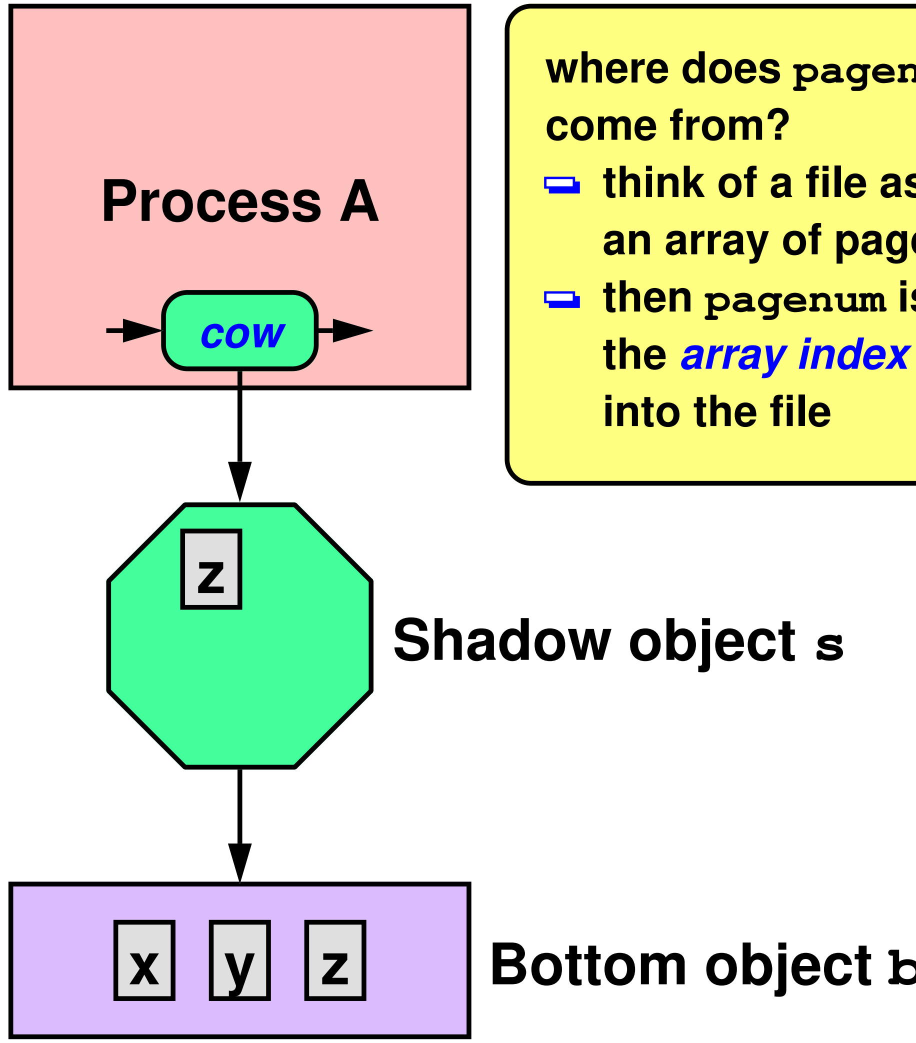
Memory Management Objects in `weenix`

- In `weenix`, an *mmobj* is used to manage *page frames*
  - types of *mmobj* in kernel assignments are:
    - there's one that lives *inside a vnode* (`vn->vn_mmobj`)
    - a *shadow object* is an mmobj
    - an *anonymous object* (meaning not associated with a file and not a shadow object) is an mmobj
  - a `vmarea` is supported by one of these 3 mmobjs



Memory Management Objects in `weenix`

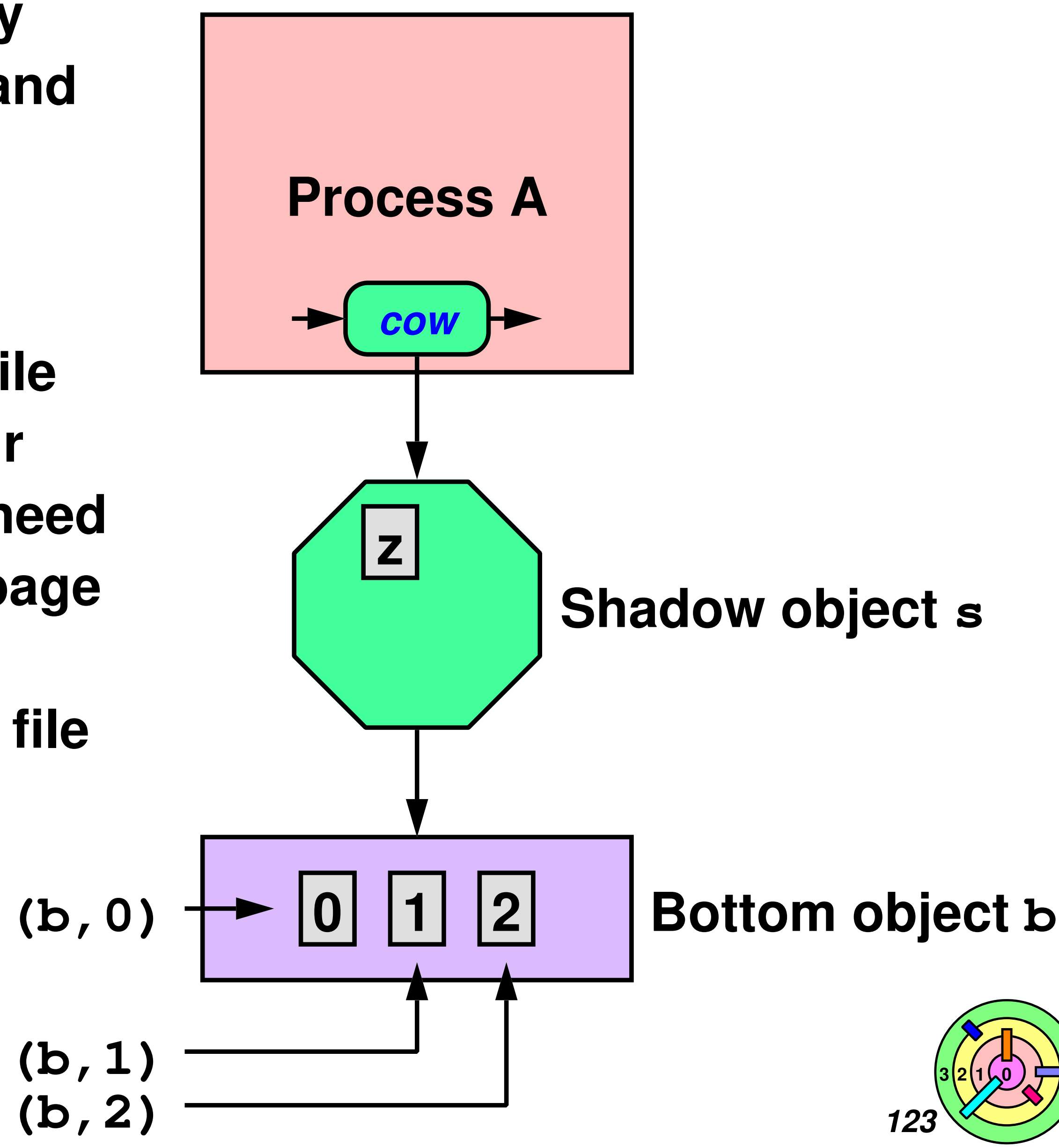
- In `weenix`, an *mmobj* is used to manage *page frames*
  - a page frame is uniquely identified by an `mmobj` and a `pagenum`
    - notation: `(mmobj, pagenum)`



where does `pagenum` come from?
- think of a file as an array of pages
- then `pagenum` is the *array index* into the file

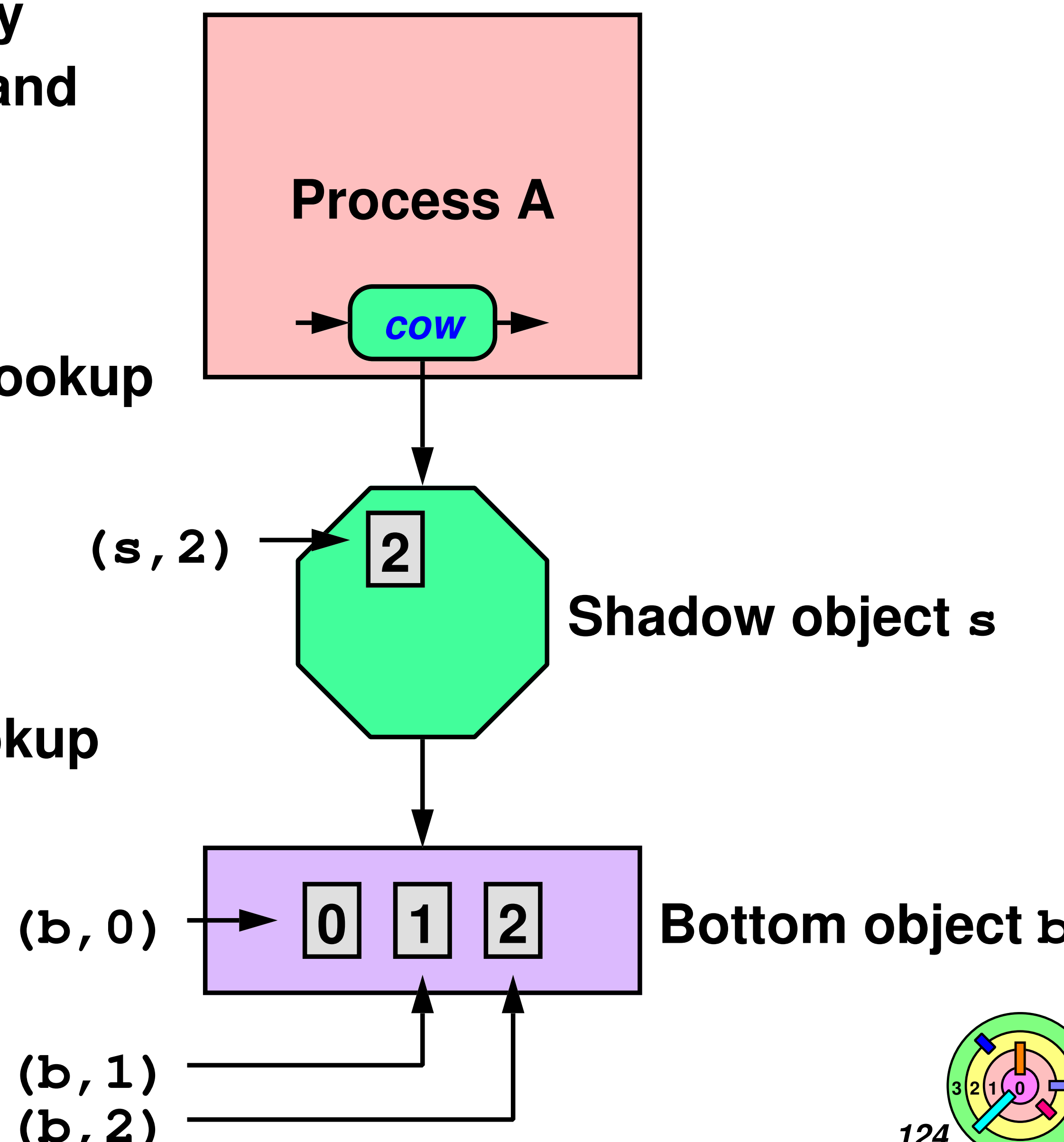
Memory Management Objects in `weenix`

- In `weenix`, an *mmobj* is used to manage *page frames*
  - a page frame is uniquely identified by an `mmobj` and a `pagenum`
    - notation: `(mmobj, pagenum)`
    - if you map part of a file (say a page) into your address space, you need to remember which page
      - `pagenum` is then a *page index* in that file



Memory Management Objects in `weenix`

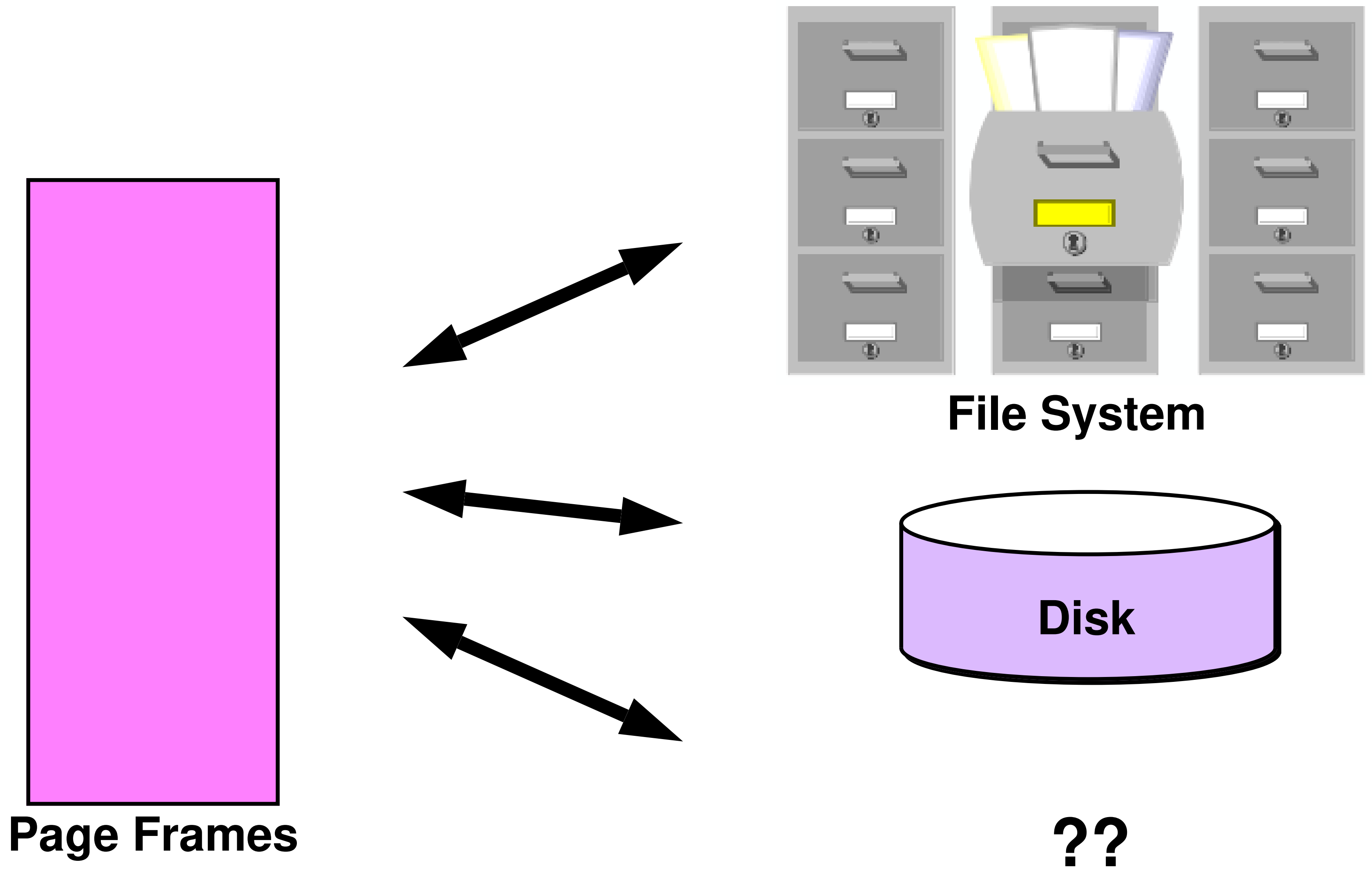
- In `weenix`, an *mmobj* is used to manage *page frames*
  - a page frame is uniquely identified by an `mmobj` and a `pagenum`
    - notation: `(mmobj, pagenum)`
    - *hash table* used for lookup
    - read kernel 3 FAQ
  - sometimes, you know the exact name of a page frame
    - use hash table to lookup
  - sometimes, you only know `pagenum` (e.g., "where is page `z`?")
    - need to *search*



7.3 Operating System Issues

- General Concerns
- Representative Systems
- Copy on Write and Fork
- *Backing Store Issues*

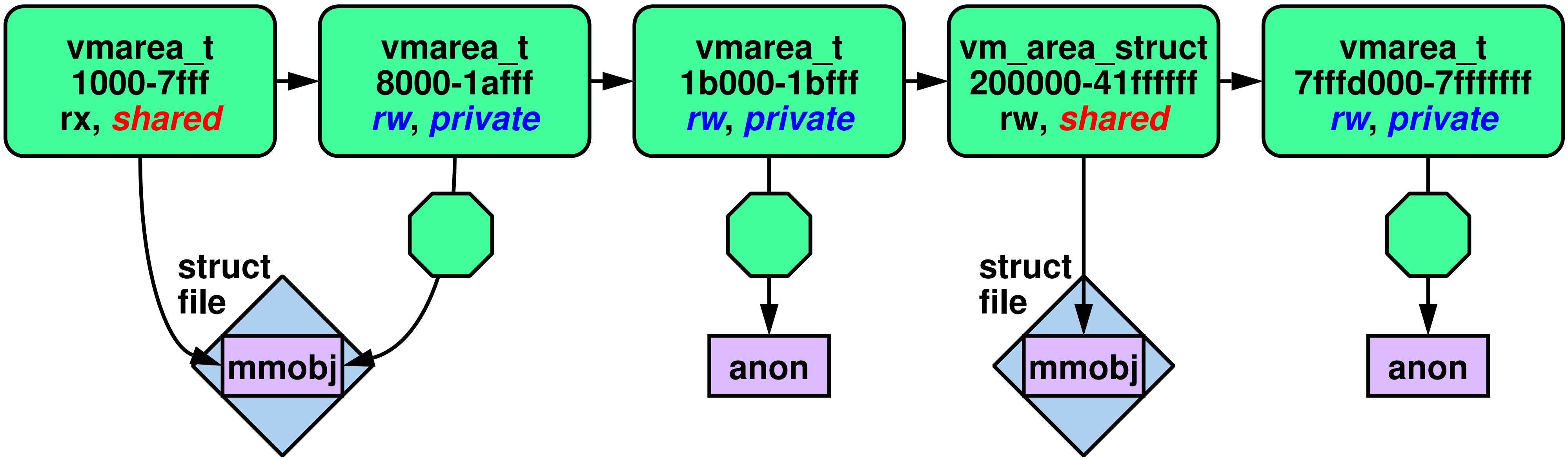
The Backing Store



Backing Up Pages (1)

- *Read-only* mapping of a file (e.g. text)
  - pages come from the file, but, since they are never modified, they never need to be written back

- Read-write *shared* mapping of a file (e.g. via `mmap()` system call)
  - pages come from the file, modified pages are written back to the file



- `weenix` supports this type of "backing store"

Backing Up Pages (2)

- Read-write *private* mapping of a file (e.g. the data section as well as memory mapped private by the `mmap()` system call)
  - pages come from the file, but *modified pages*, associated with *shadow objects*, must be backed up in *swap space*

- Anonymous memory (e.g. bss, stack, and shared memory), also *privately* mapped
  - pages are created as *zero fill on demand;* they must be backed up in *swap space*
    - *modified pages* of these, associated with *shadow objects*, must be backed up in *swap space*

- `weenix` does *not* support this type of backing store
  - need to prevent the pageout daemon to free up these pages accidentally
    - simply move them out of the pageout daemon's way using `pframe_pin()`

Swap Space

- Swap space management possibilities
  - radically-*conservative* approach: *Eager Evaluation* (or pre-allocation)
    - backing-store space is allocated when virtual memory is allocated
    - page outs always succeed
    - disadvantage: might need to have much more backing store than needed
  - radically-*liberal* approach: *Lazy Evaluation*
    - backing-store space is allocated only when needed
    - advantage: can get by with minimal backing-store space
    - disadvantage: page outs could fail because of no space and process gets killed at a seemingly random time

Swap Space

- Space management possibilities
  - mixed approach: e.g., reserve stack space for a thread in Windows
    - the address space for the thread stack is first *"reserved"*
      - no backing store actually created, but space is reserved so no other thread can use the reserved space
    - when part of this address space is used, it's *"committed"* (backing store is actually allocated)

- For things like `malloc()` and allocation of address space for privately-mapped files
  - by default, done with eager evaluation in Windows and most Unix/Linux systems
  - both systems provide means for lazy evaluation as well

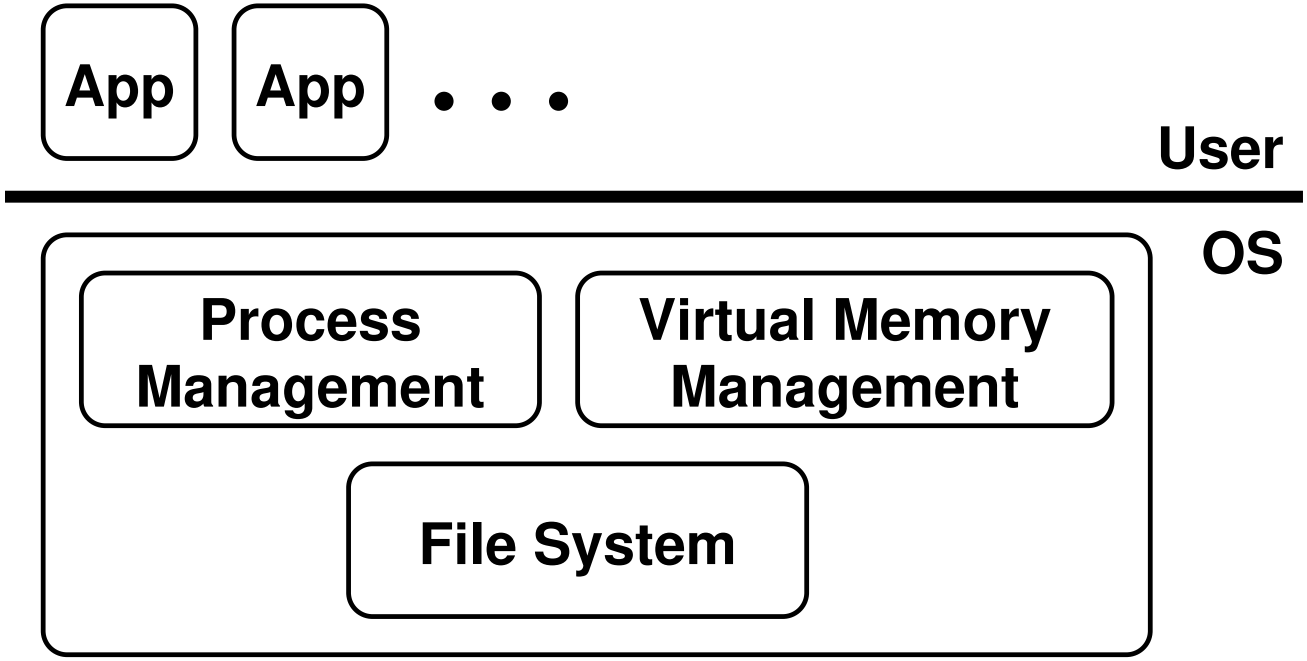
Space Allocation in Linux

- Total memory = primary + swap space
- System-wide parameter: overcommit_memory
  - three possibilities
    - maybe (default)
    - always
    - never
- mmap has MAP_NORESERVE flag
  - don't worry about over-committing

Space Allocation in Windows

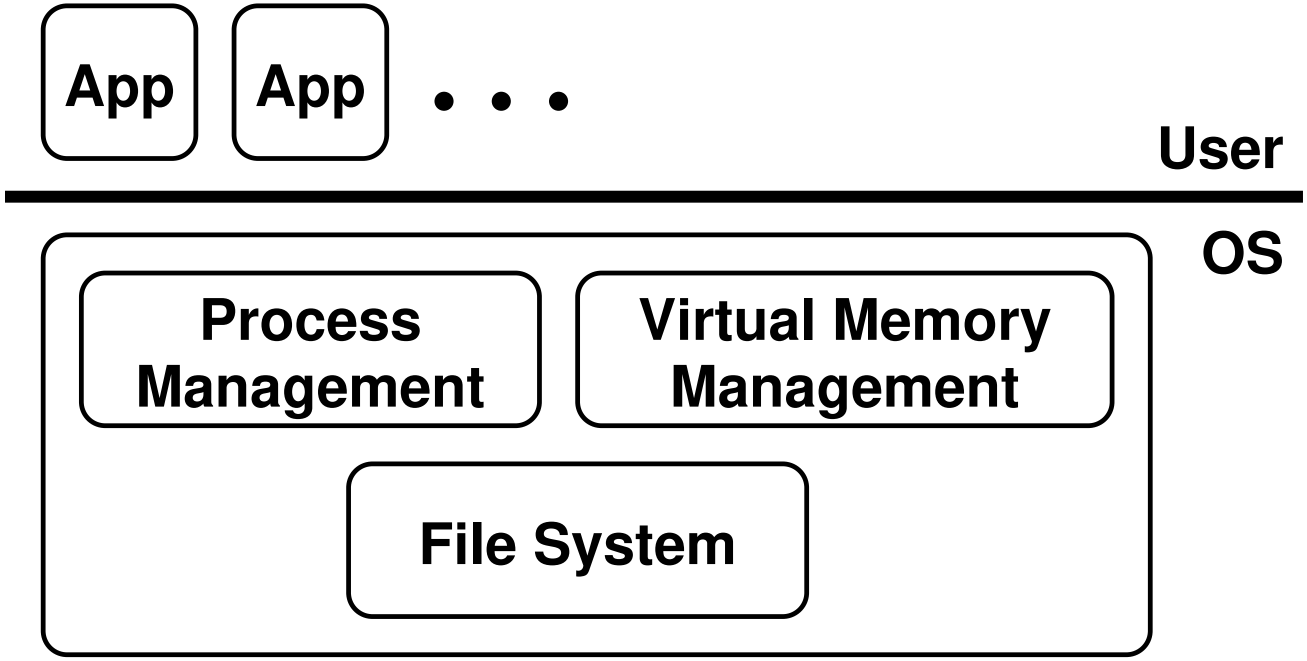
- Space reservation
  - allocation of virtual memory
- Space commitment
  - reservation of physical resources
    - paging space + physical memory
- `MapViewOfFile` (sort of like `mmap`)
  - no over-commitment
- Thread creation
  - creator specifies both reservation and commitment for stack pages

Summary



- The subsystems are inter-related
  - file systems uses threads managed by the process subsystem
  - file systems uses buffer cache (managed by the memory subsystem)
  - memory subsystem uses threads to do background work
  - process subsystem keeps track of data structures related to files and virtual memory on behalf of processes

Summary



- To make sure you understand the big picuture
  - think of everything that happens in these subsystems when you type "`ls`" into a console
- Kernel 3 is where everything comes together
  - although we are already using page tables in earlier assignments (see `pt_init()`)