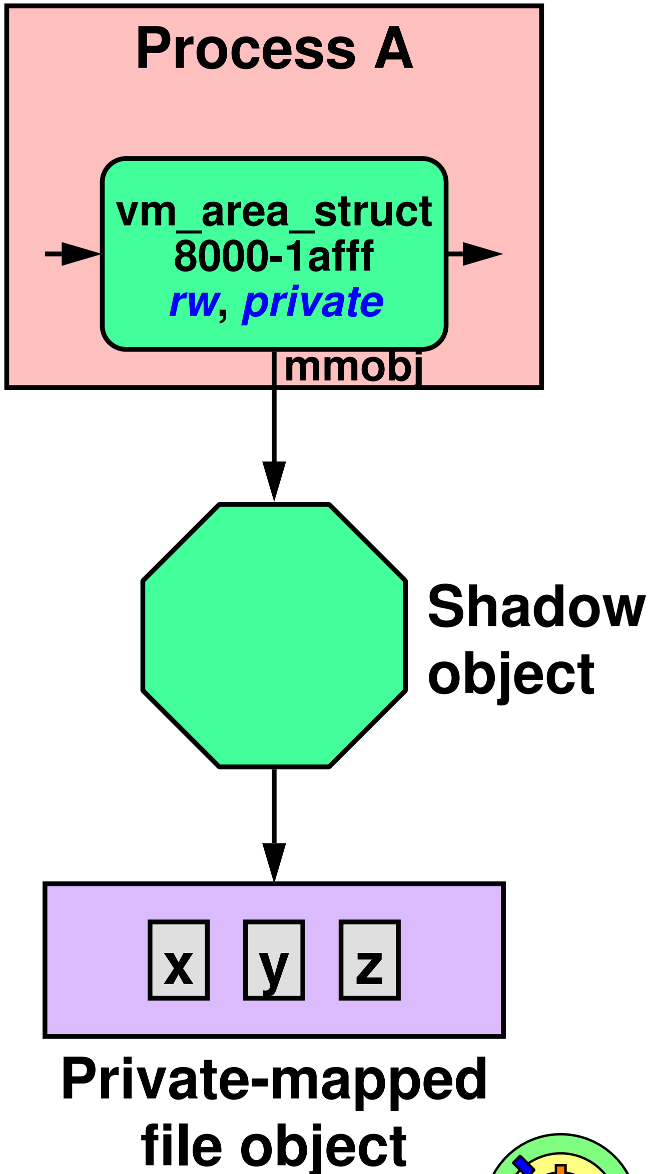
Copy-on-write & Fork

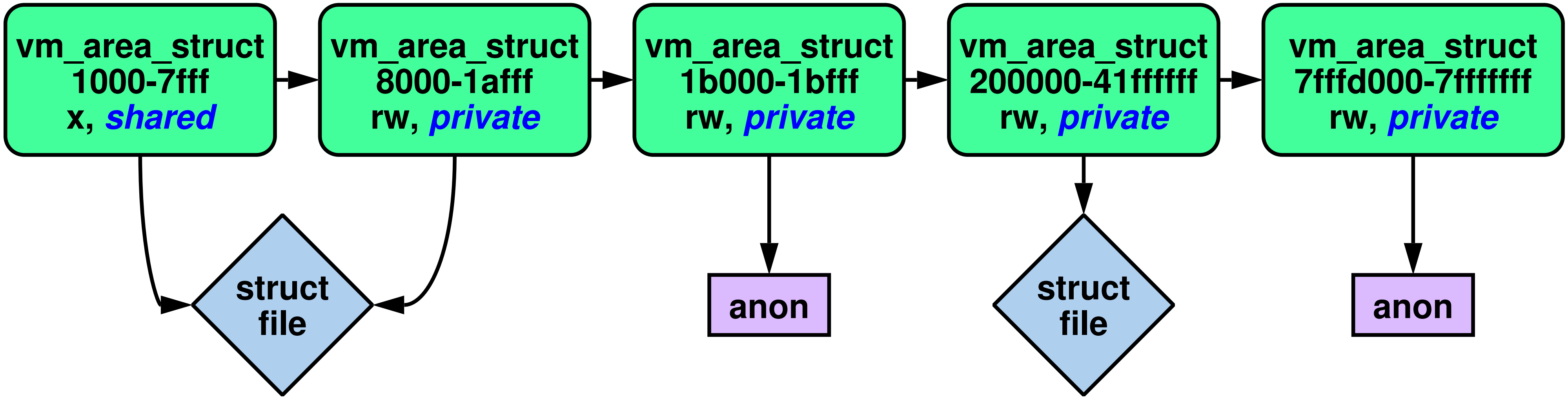
- *Shadow Objects*
  - indirection
  - keep track of pages that were *originally* *copy-on-write* but have been *modified*
- A page in a memory map, into which an object was mapped *private* (e.g., data region), has an associated *shadow object*
  - if a page is "managed by a shadow object" (or "referenced 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*



Copyright © William C. Cheng

Address-Space Representation

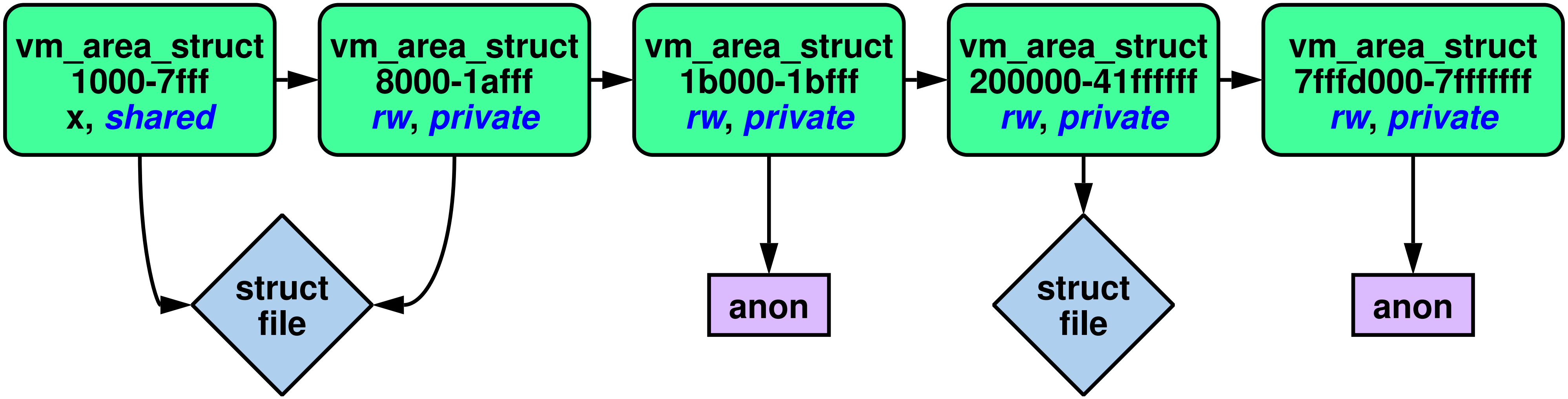
- Remember this?



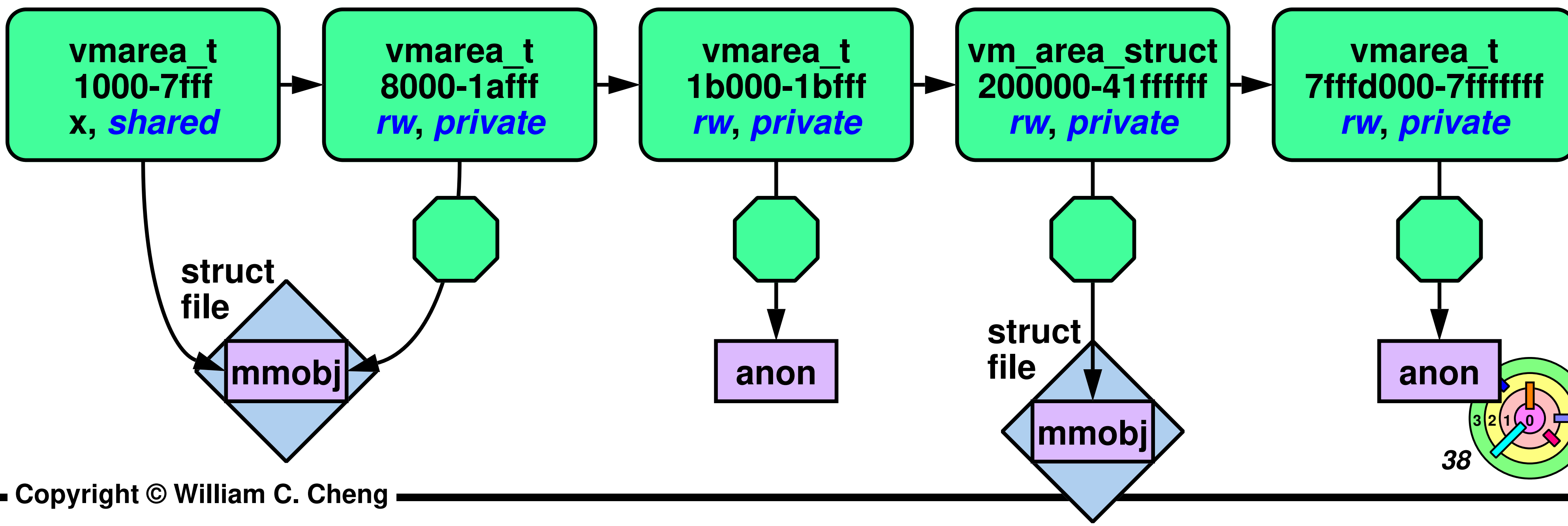
Copyright © William C. Cheng

Address-Space Representation

- Remember this?

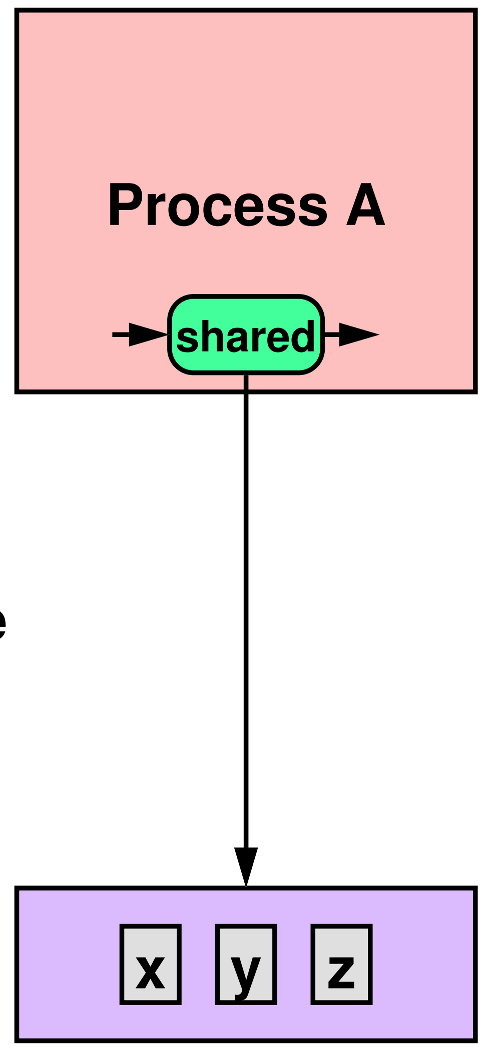


- now we have to start with (`mmobj` is used in `weenix`):



Copyright © William C. Cheng

Share Mapping (1)



File object

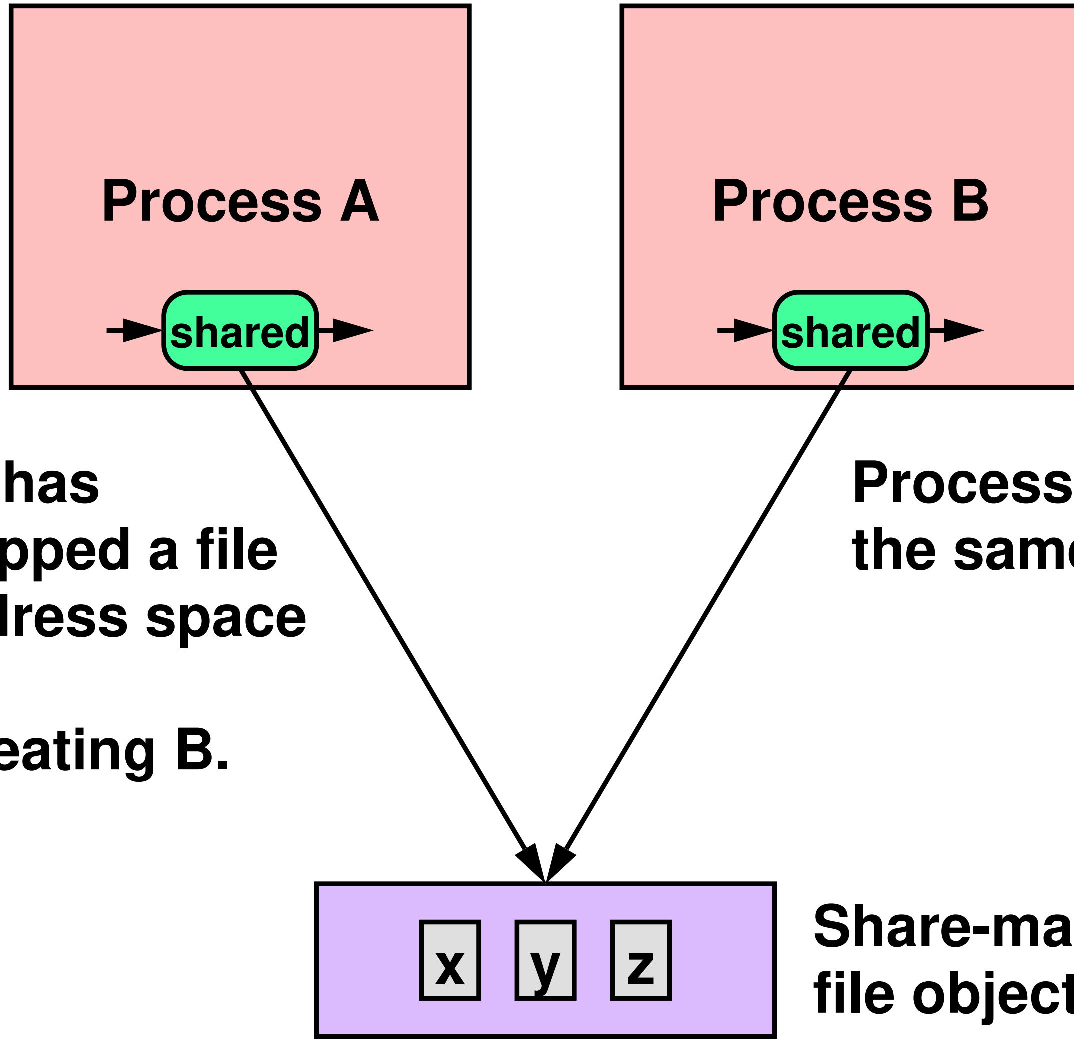
Process A has shared-mapped a file into its address space

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

Copyright © William C. Cheng

Share Mapping (2)



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

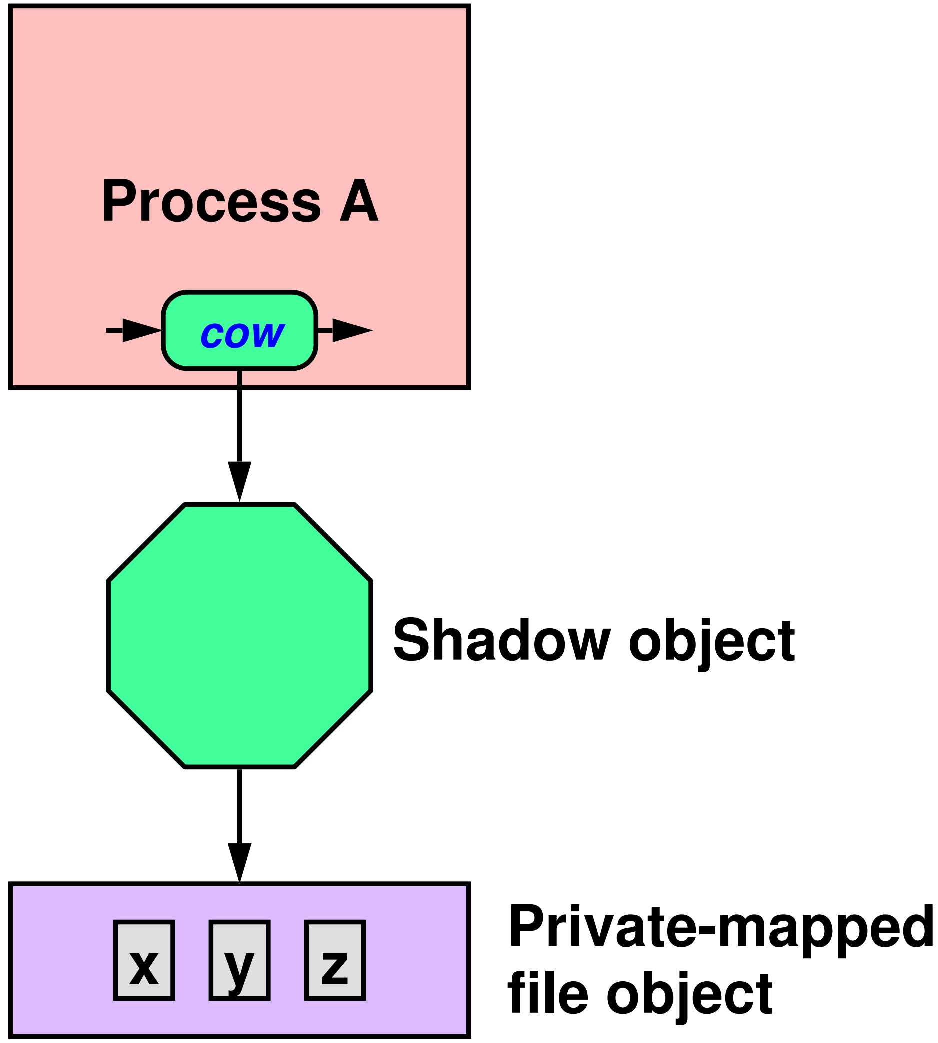
A forks, creating B.

Process B also has the same shared mapping

Share-mapped file object

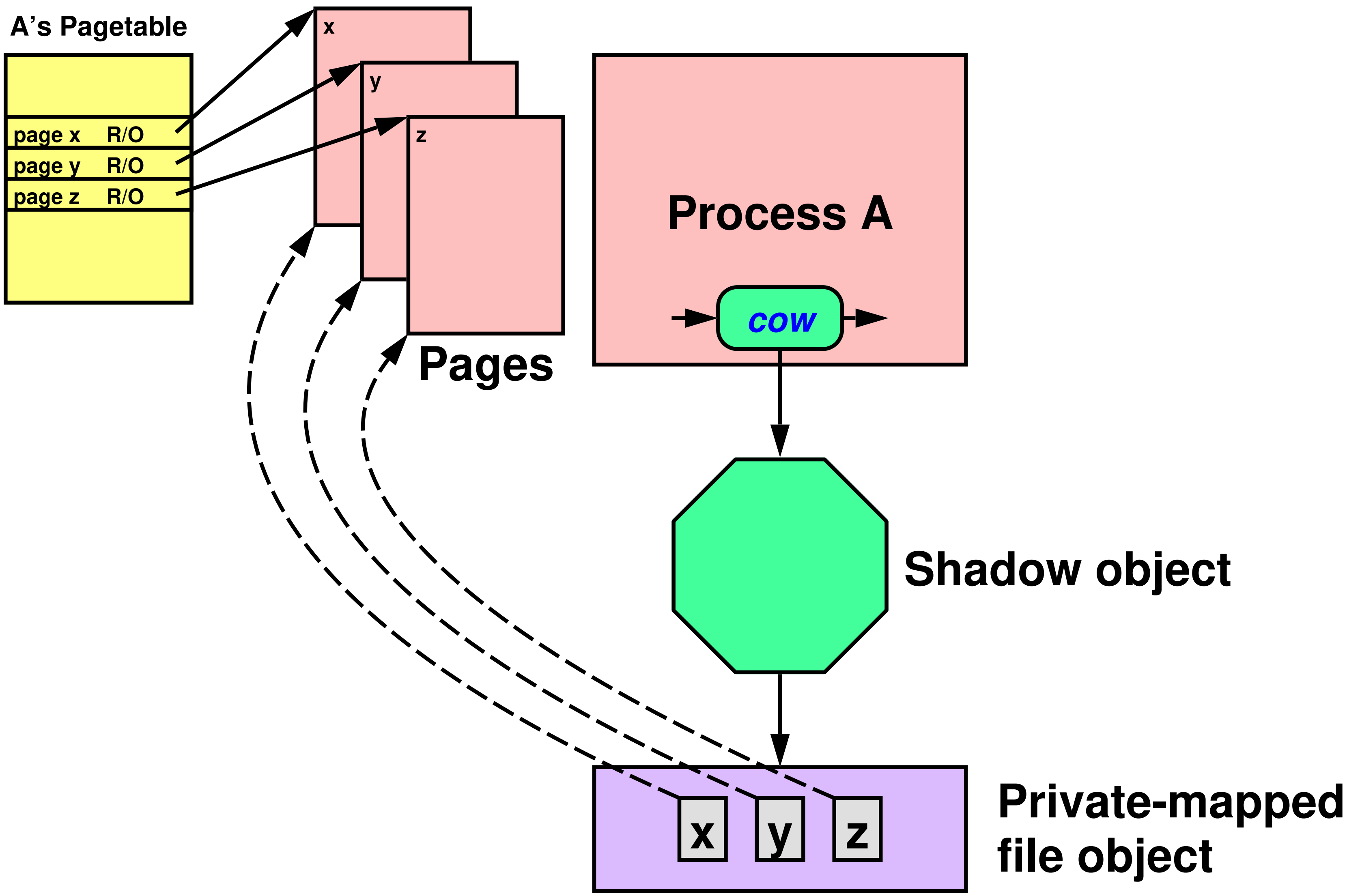
Copyright © William C. Cheng

Private Mapping (1)



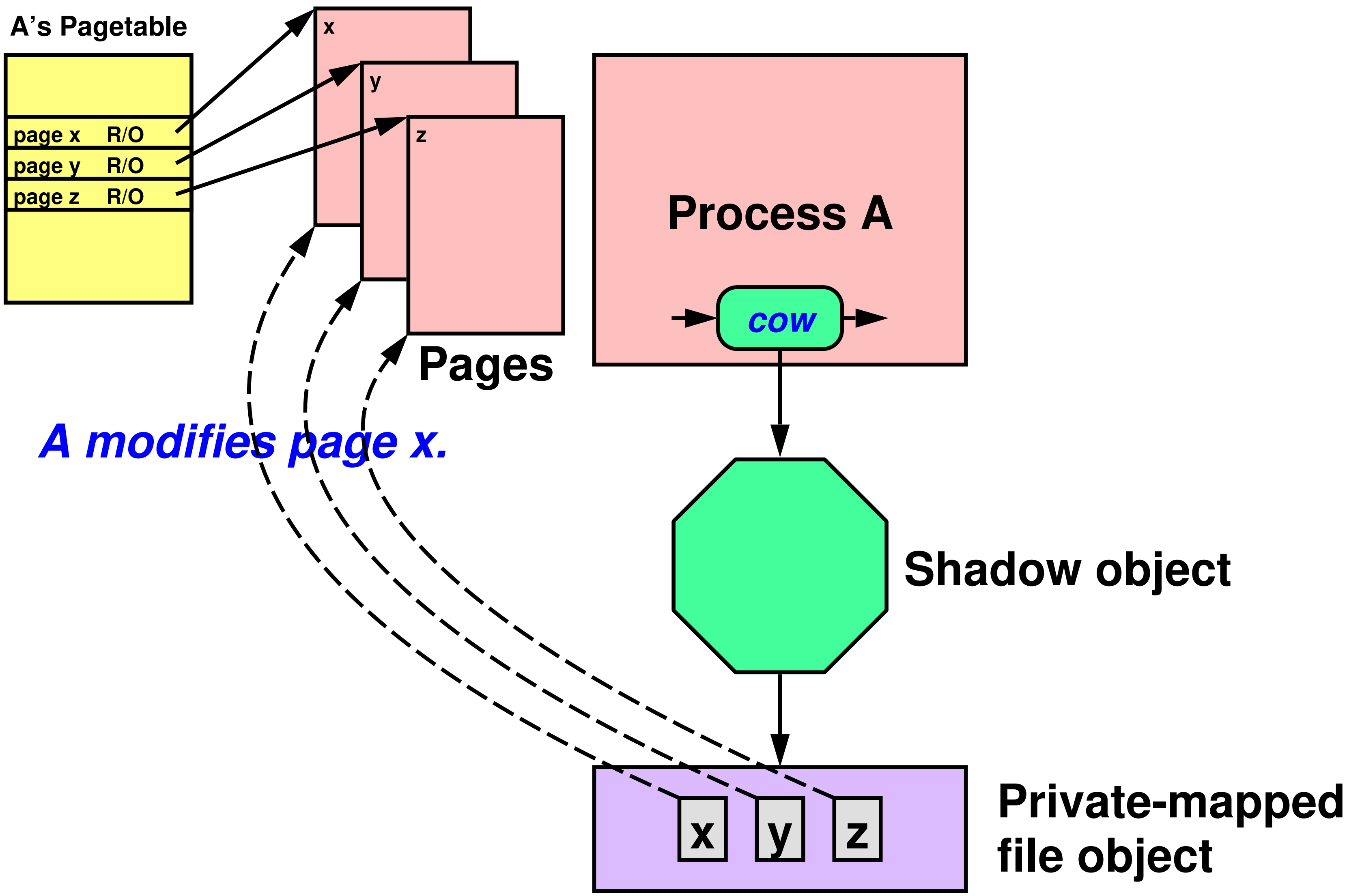
Copyright © William C. Cheng

Private Mapping (1)



Copyright © William C. Cheng

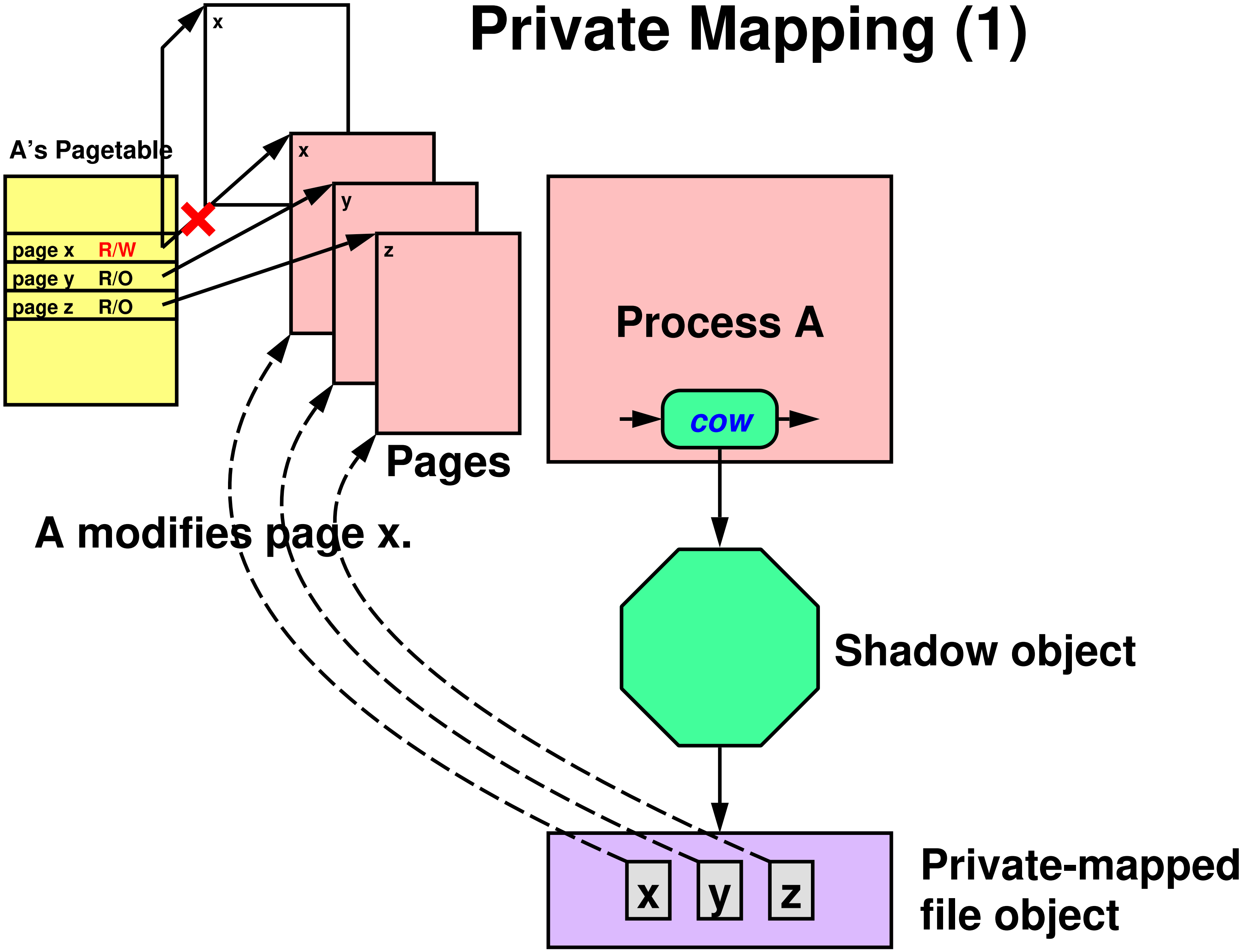
Private Mapping (1)



A modifies page x.

Copyright © William C. Cheng

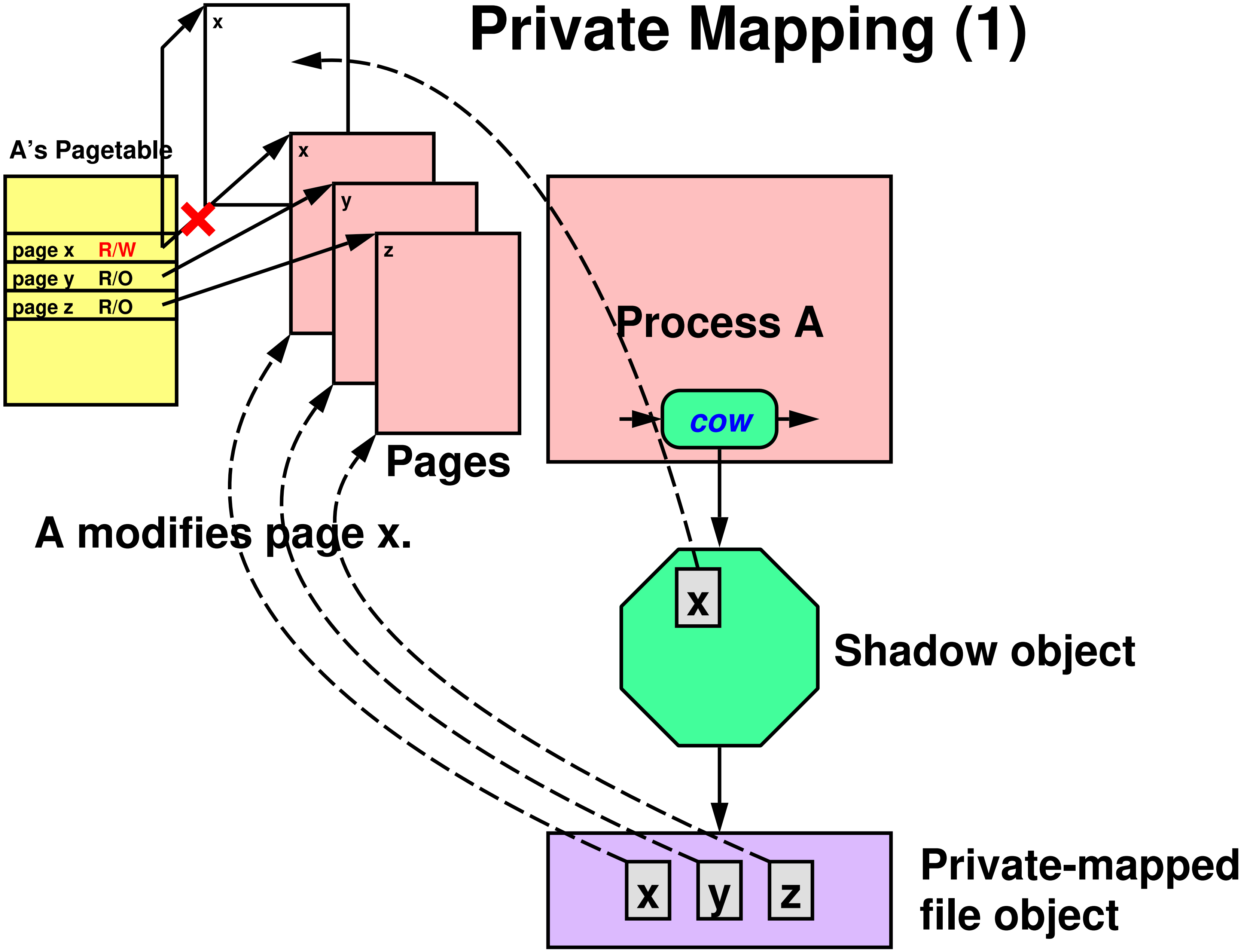
Private Mapping (1)



A modifies page x.

Copyright © William C. Cheng

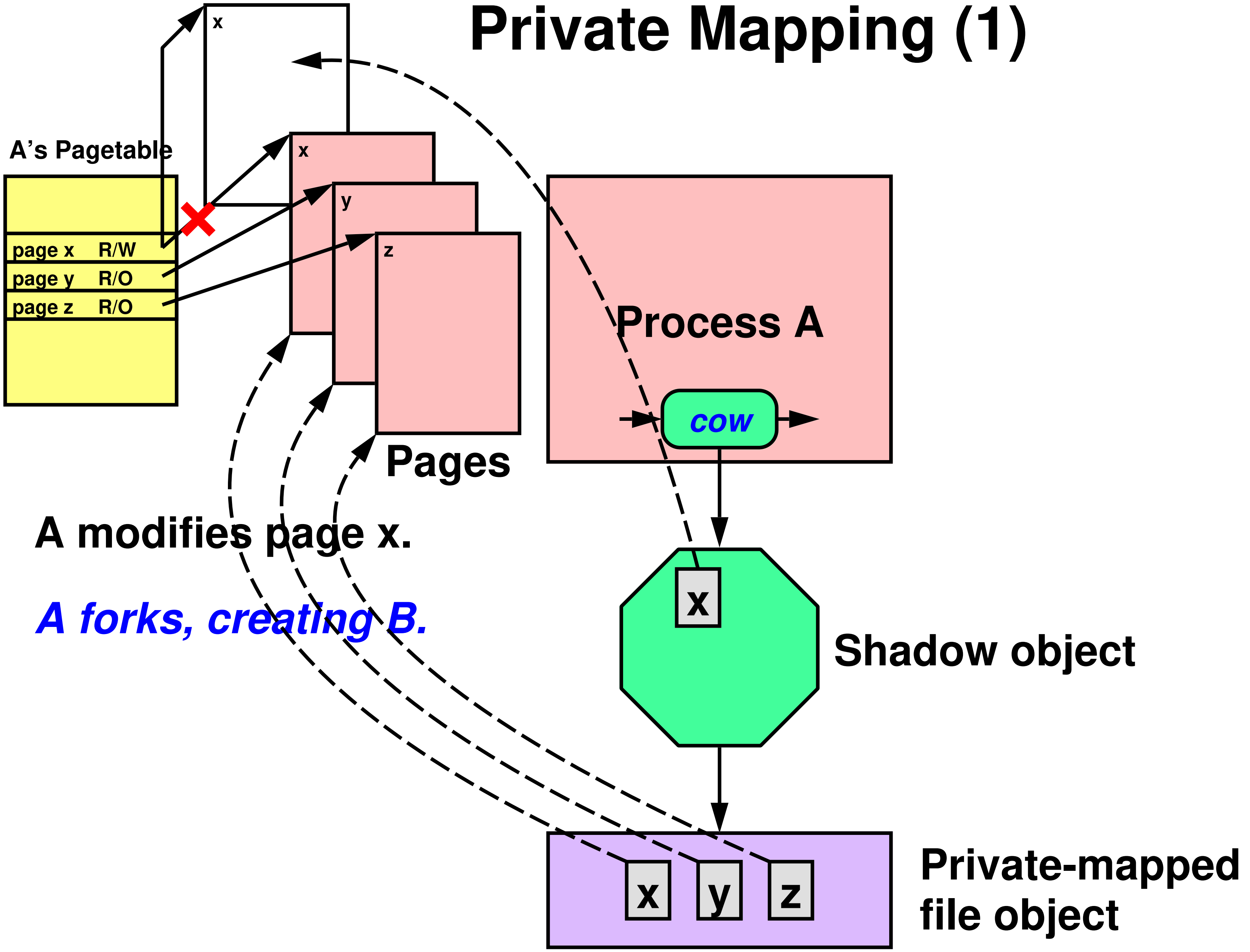
Private Mapping (1)



A modifies page x.

Copyright © William C. Cheng

Private Mapping (1)

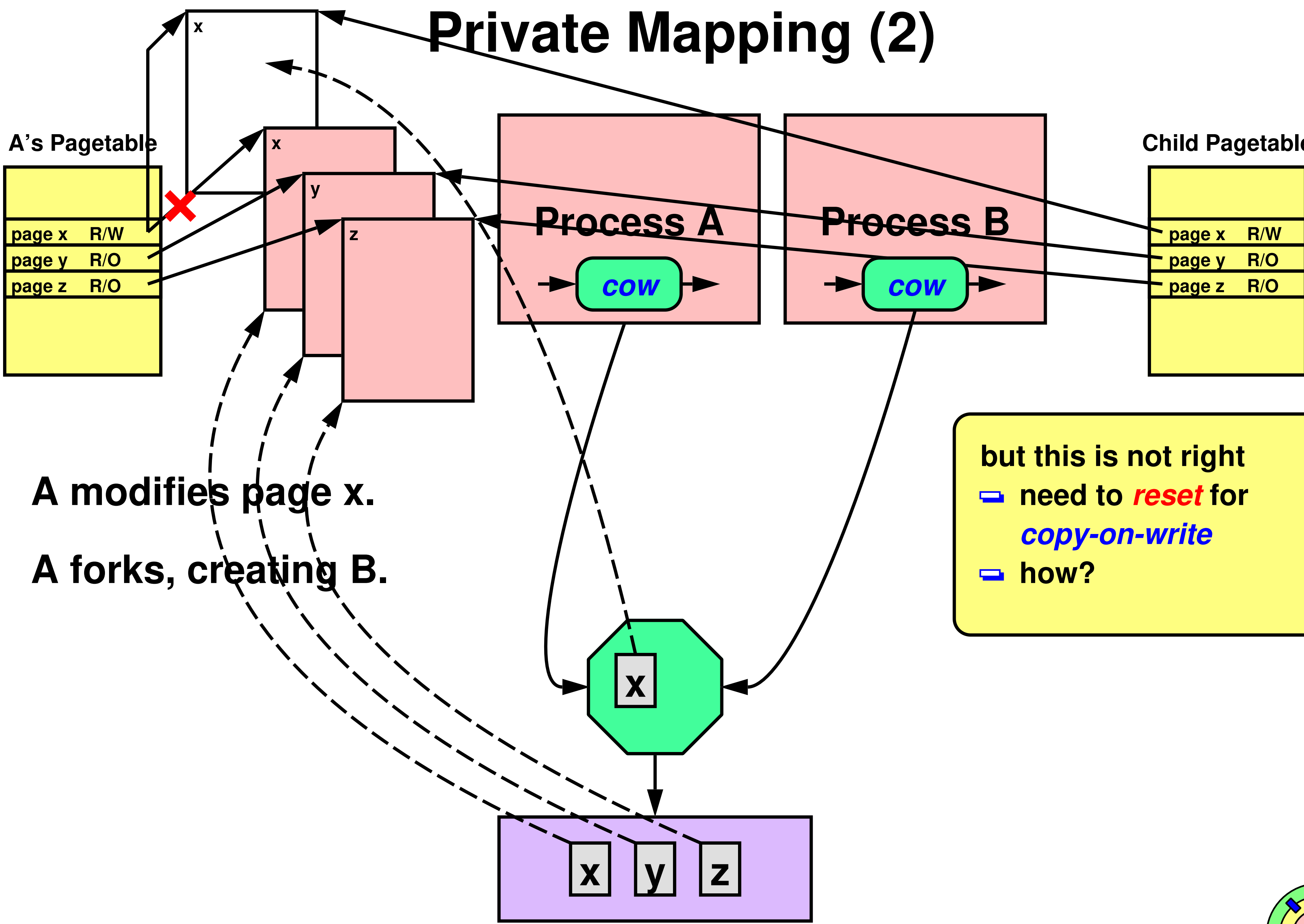


A modifies page x.

A forks, creating B.

Copyright © William C. Cheng

Private Mapping (2)



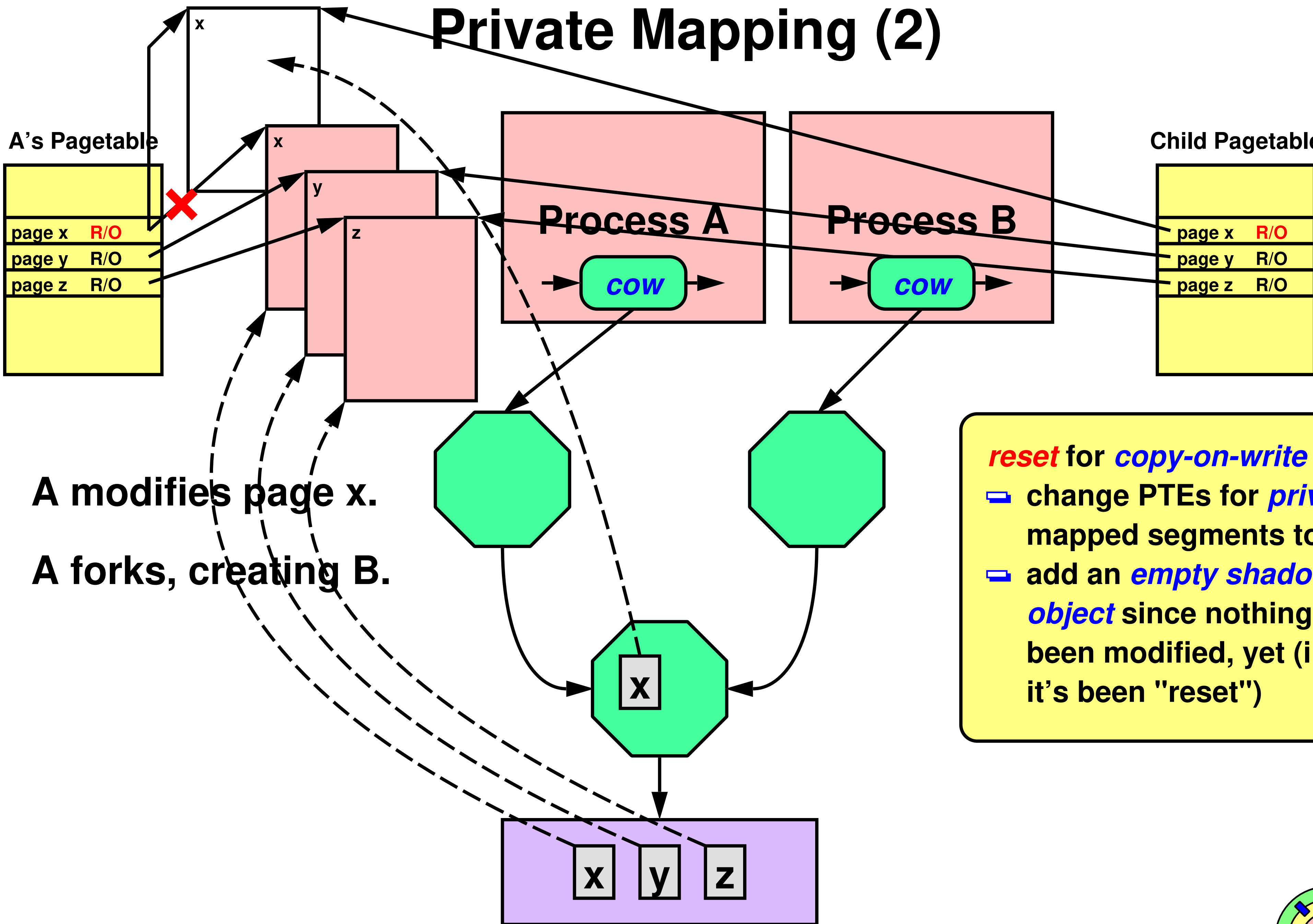
A modifies page x.

A forks, creating B.

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

Copyright © William C. Cheng

Private Mapping (2)



A modifies page x.

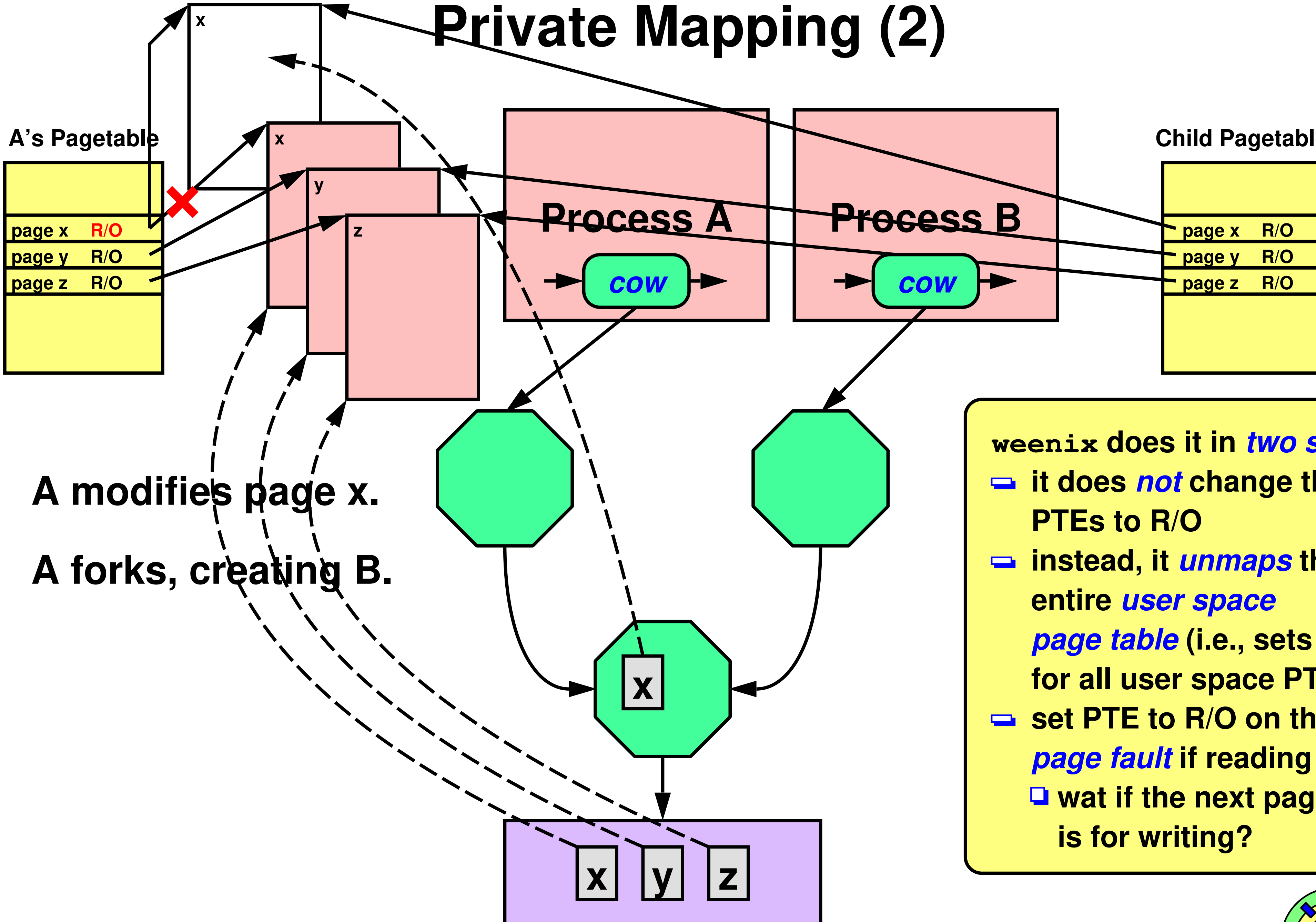
A forks, creating B.

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")

Copyright © William C. Cheng

Private Mapping (2)



A modifies page x.

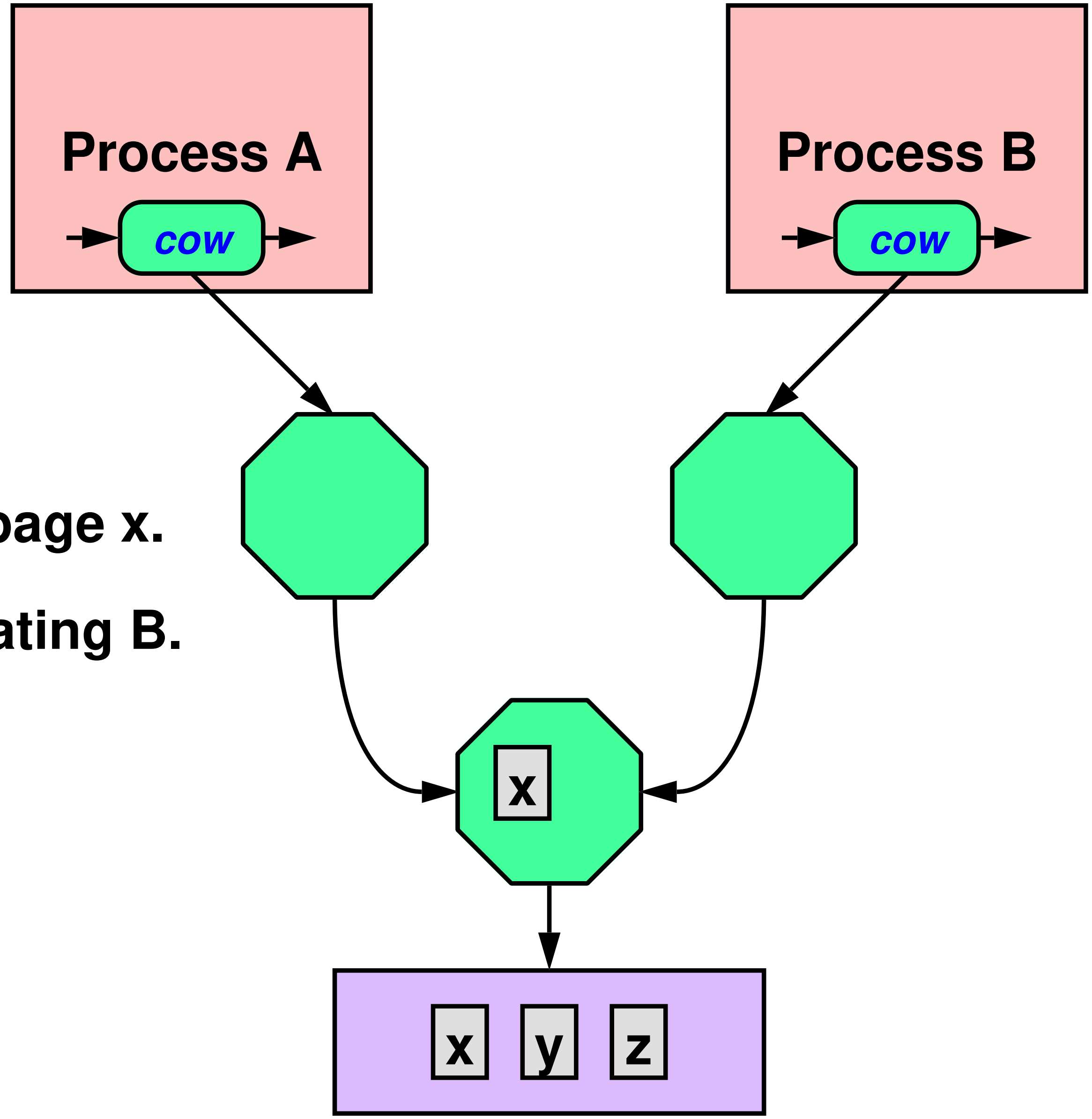
A forks, creating B.

`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?

Copyright © William C. Cheng

Private Mapping (2)

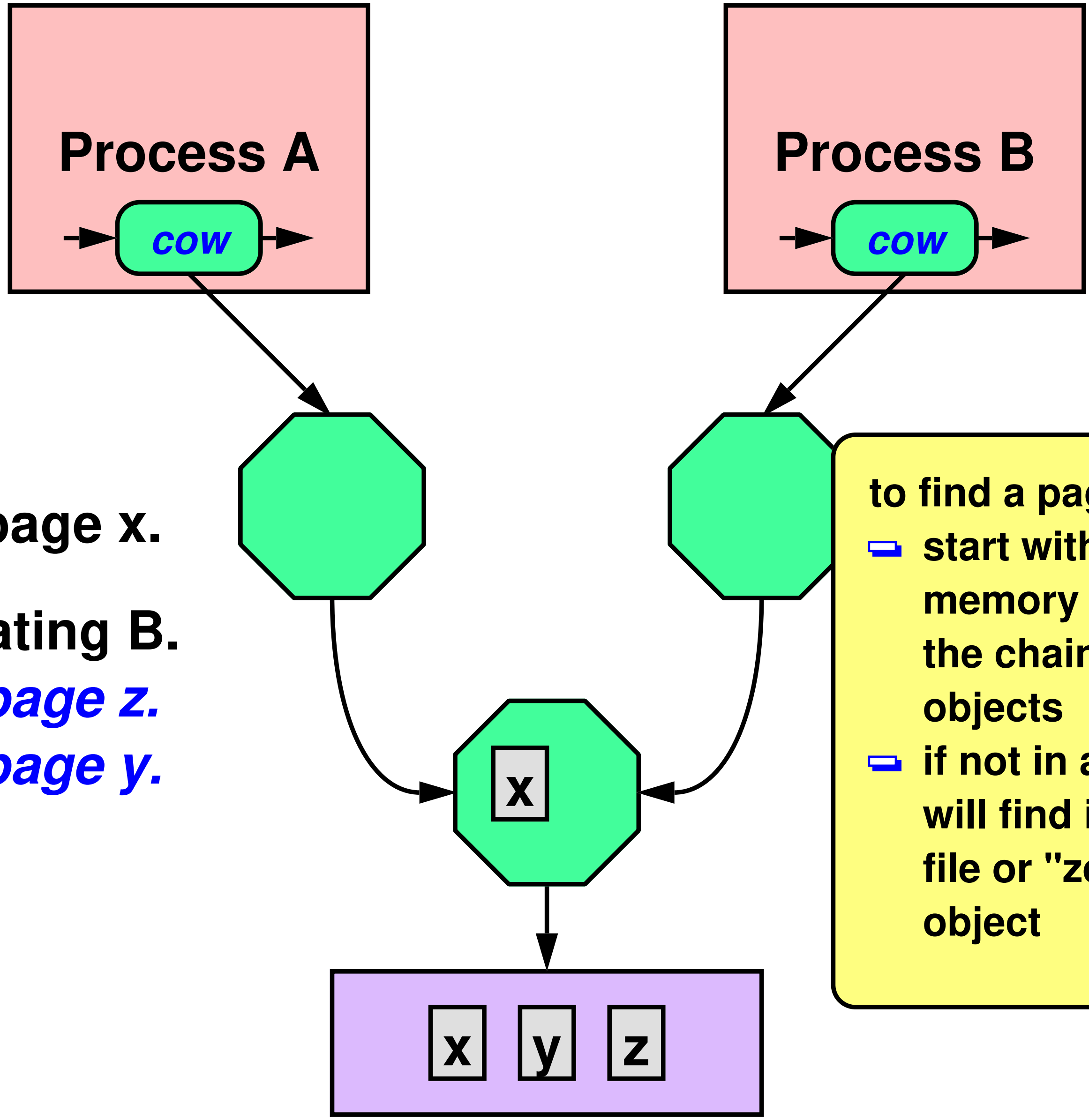


A modifies page x.

A forks, creating B.

Copyright © William C. Cheng

Private Mapping (2)



A modifies page x.

A forks, creating B.

A modifies page z.

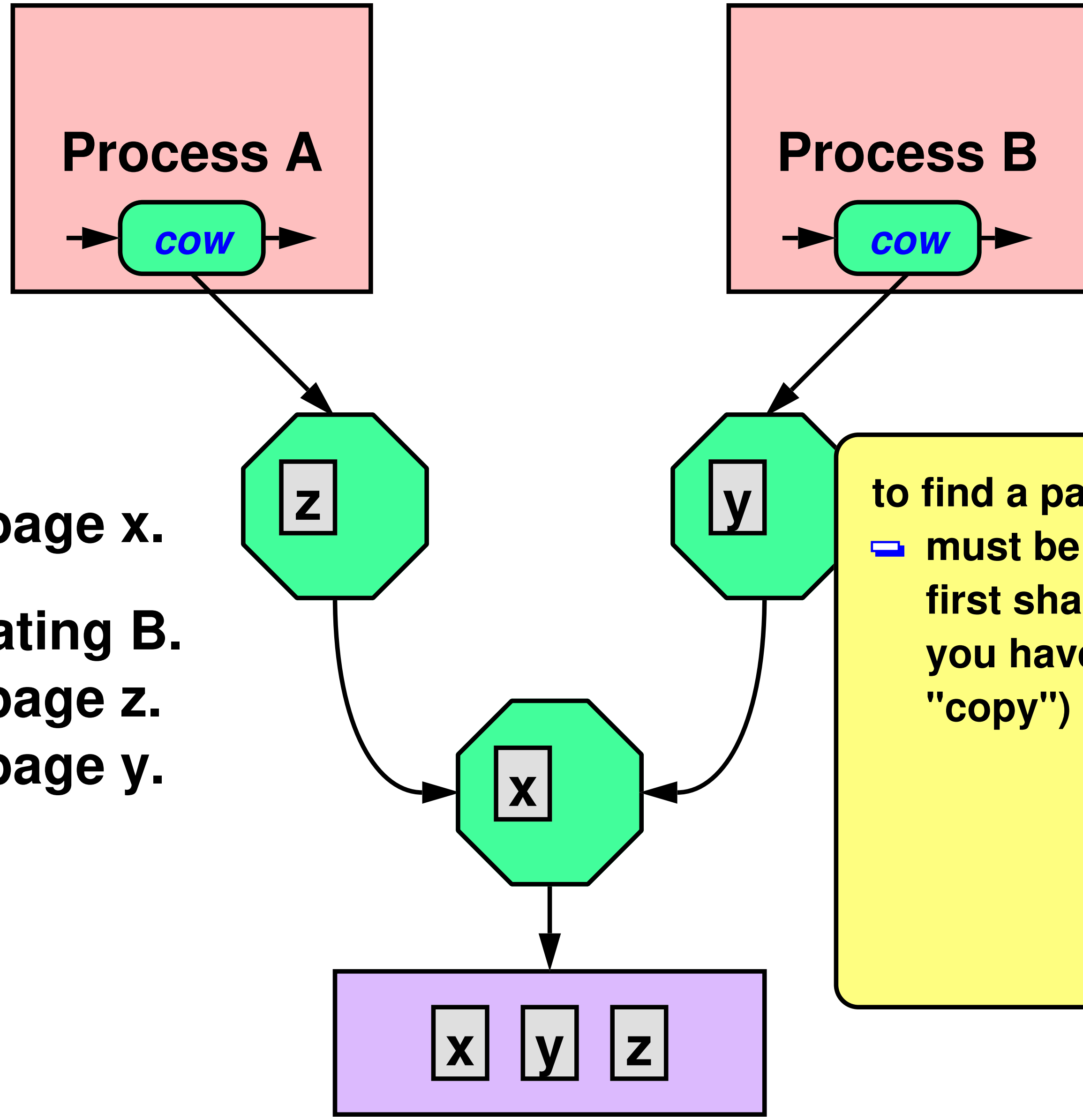
B modifies page y.

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

Copyright © William C. Cheng

Private Mapping (2)



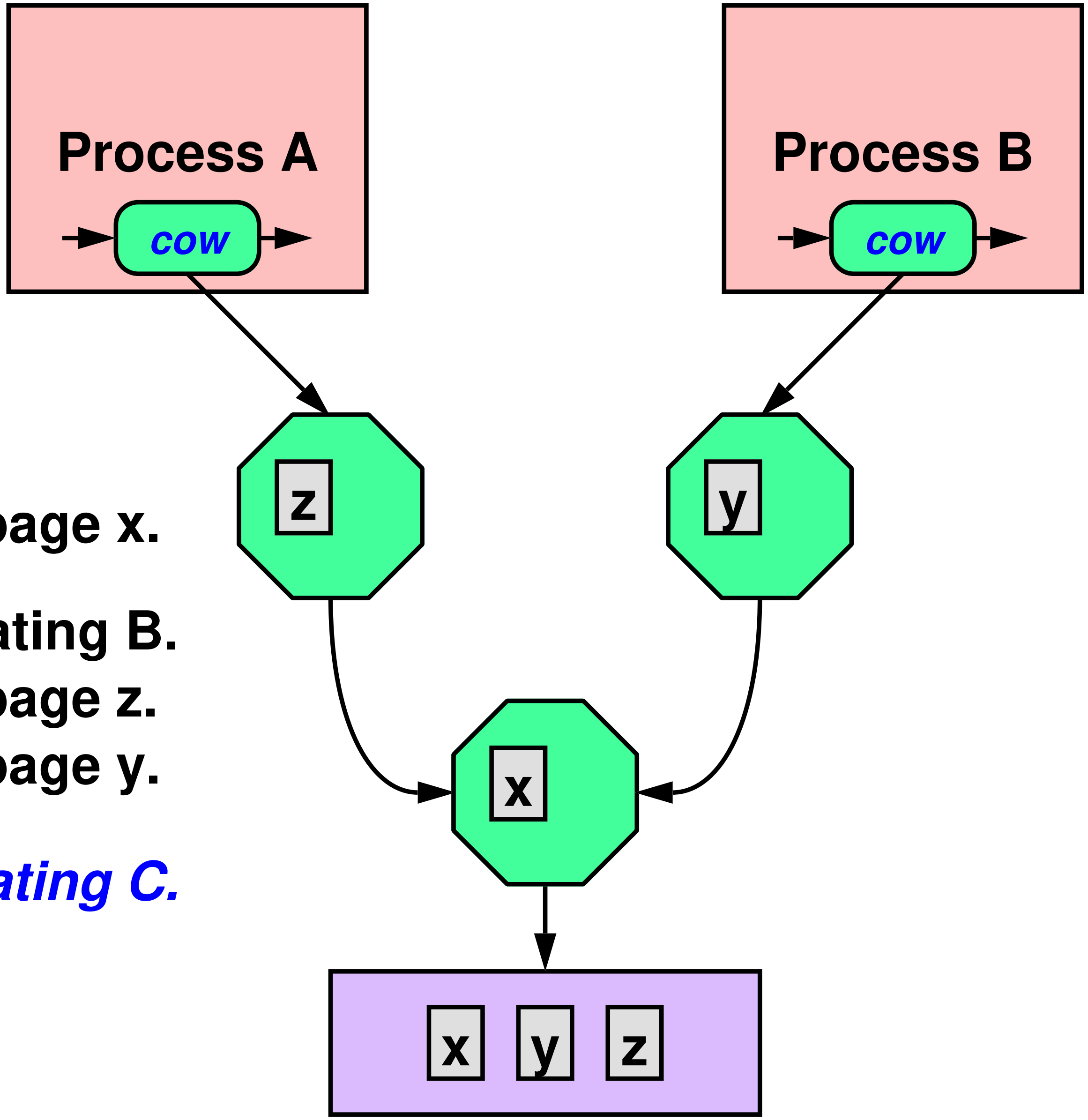
A modifies page x.

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")

Copyright © William C. Cheng

Private Mapping (2)



A modifies page x.

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

B forks, creating C.

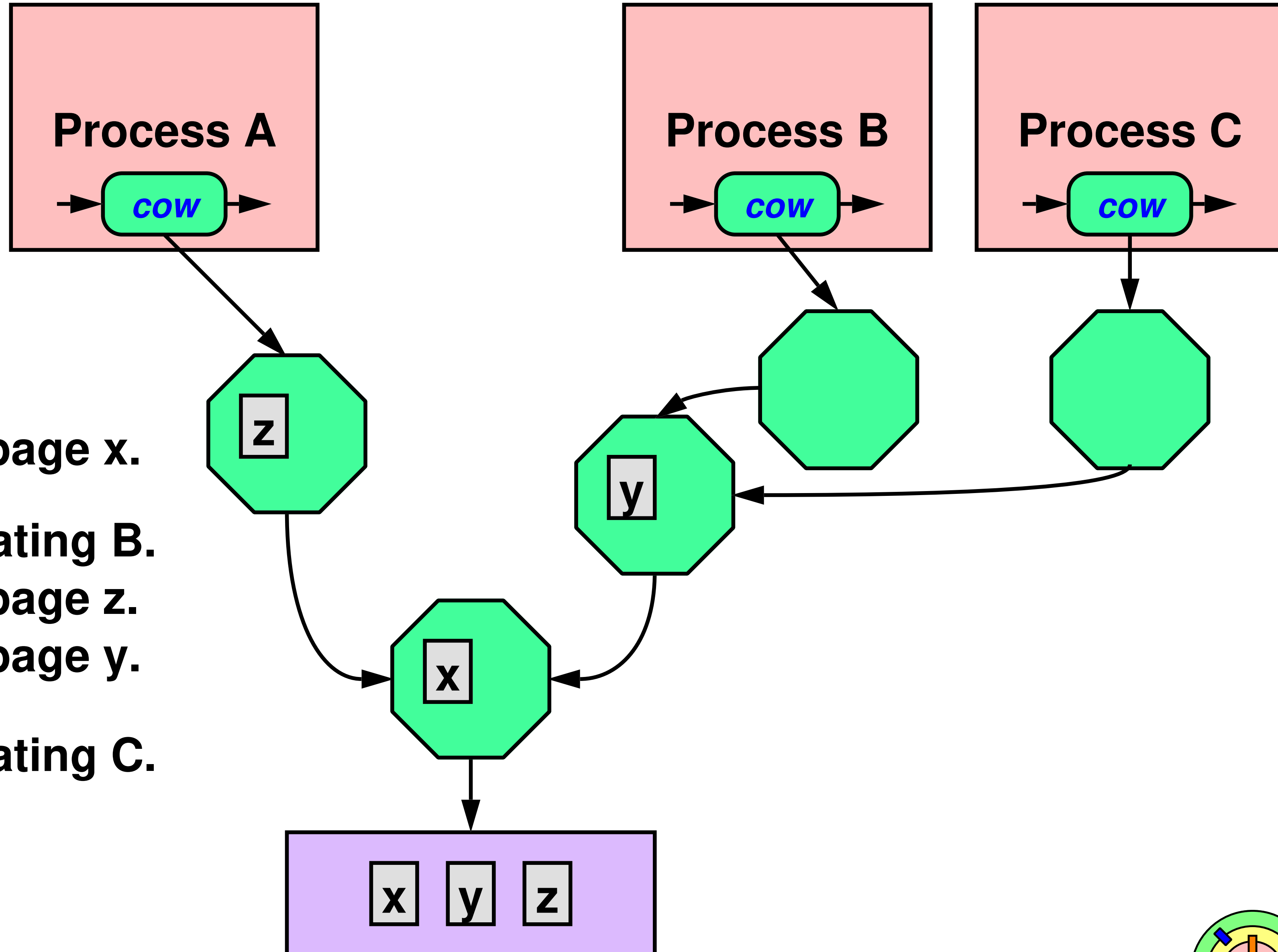
Copyright © William C. Cheng

Private Mapping (3)

A modifies page x.

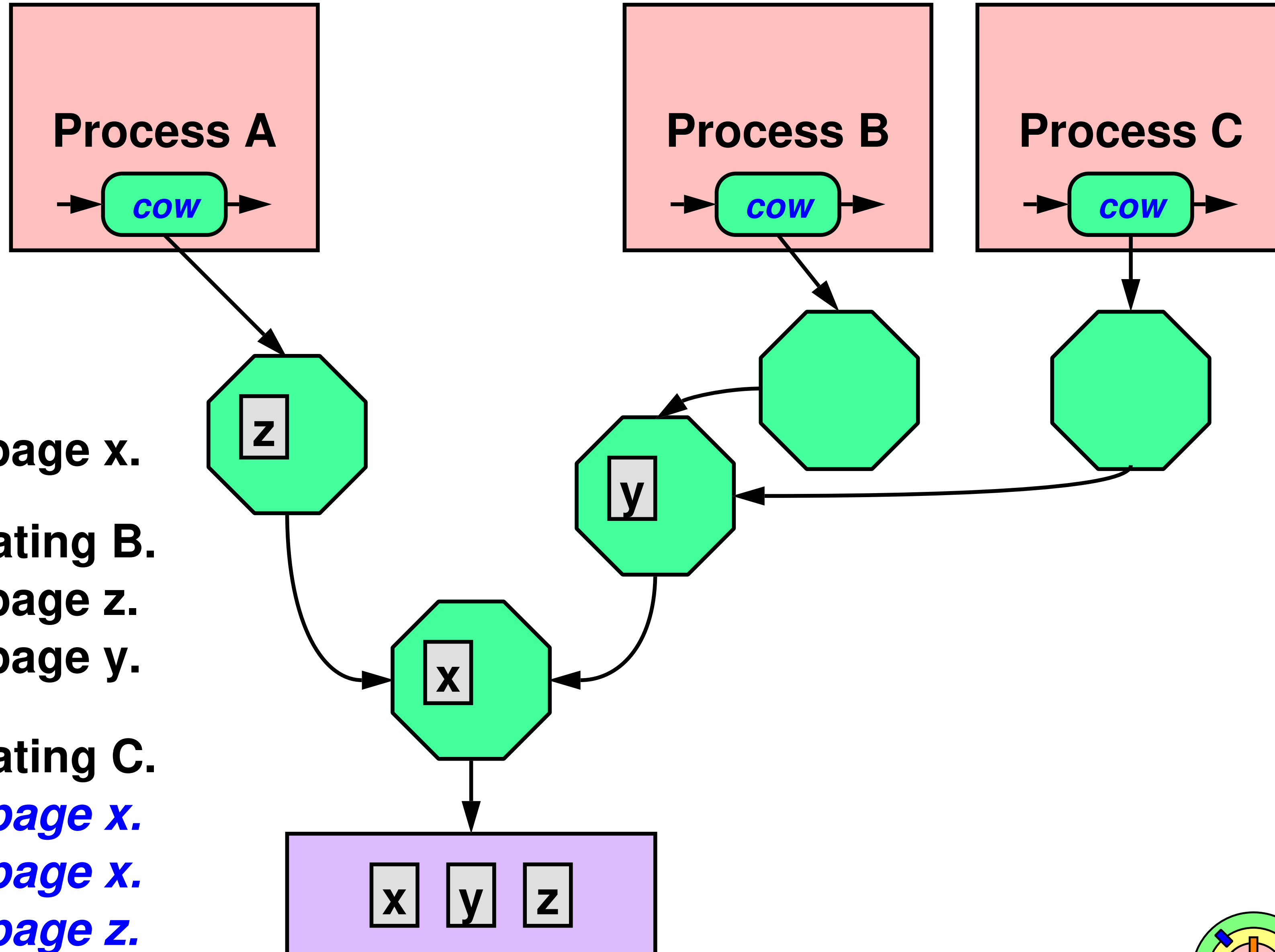
A forks, creating B.
A modifies page z.
B modifies page y.

B forks, creating C.



Copyright © William C. Cheng

Private Mapping (3)



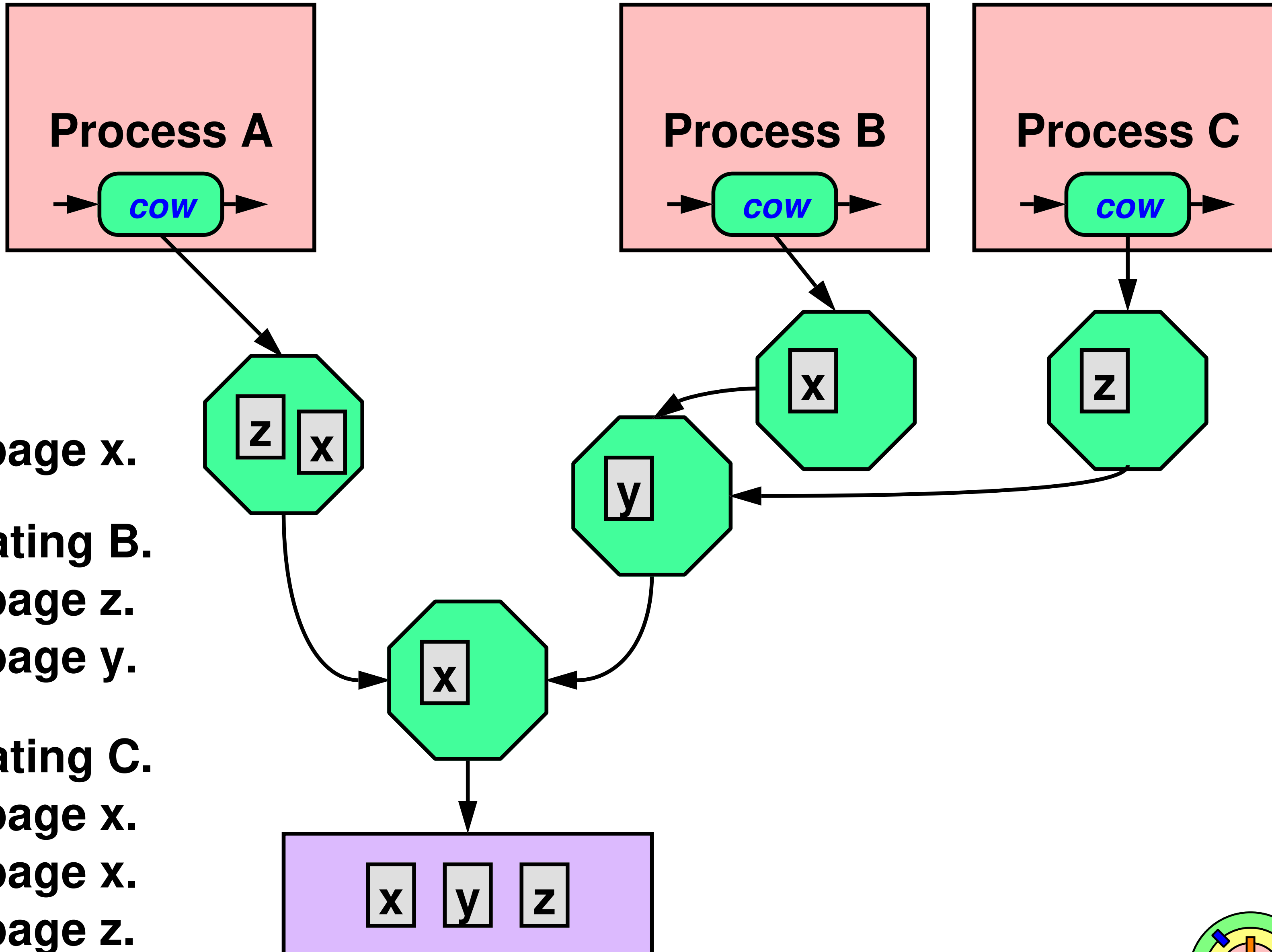
A modifies page x.

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.

Copyright © William C. Cheng

Private Mapping (3)



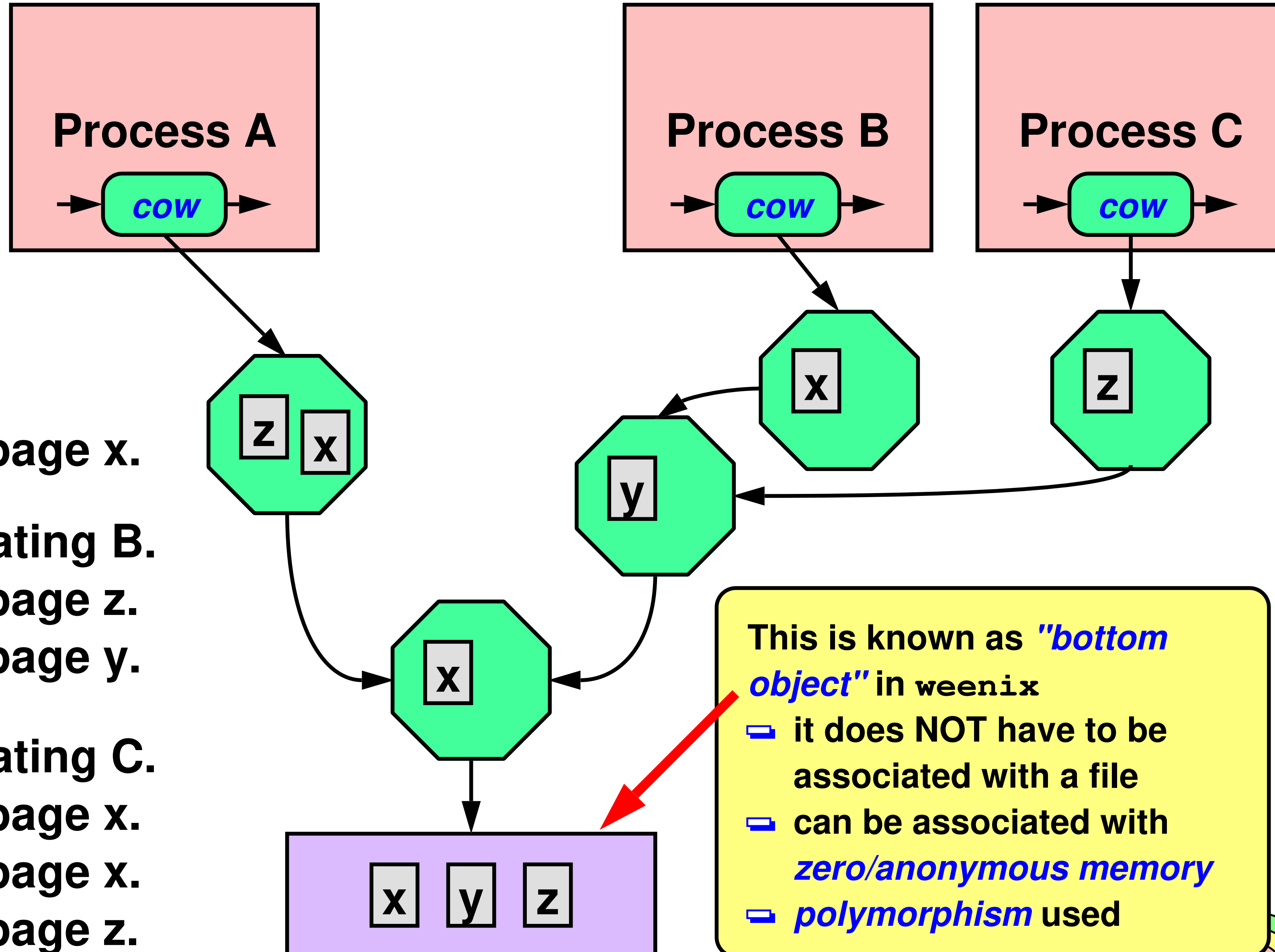
A modifies page x.

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.

Copyright © William C. Cheng

Private Mapping (3)



A modifies page x.

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

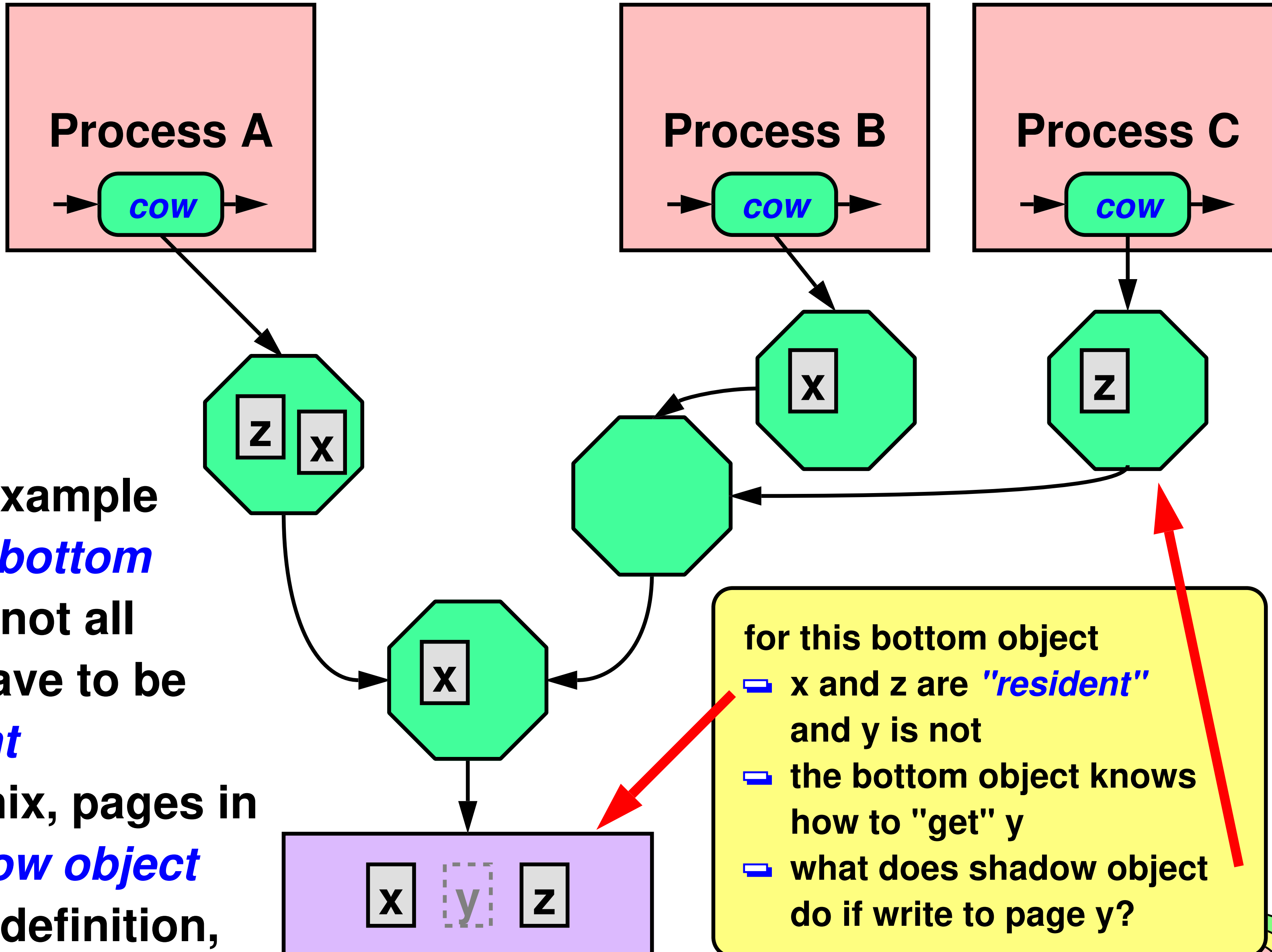
B forks, creating C.
A modifies page x.
B modifies page x.
C modifies page z.

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

Copyright © William C. Cheng

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?

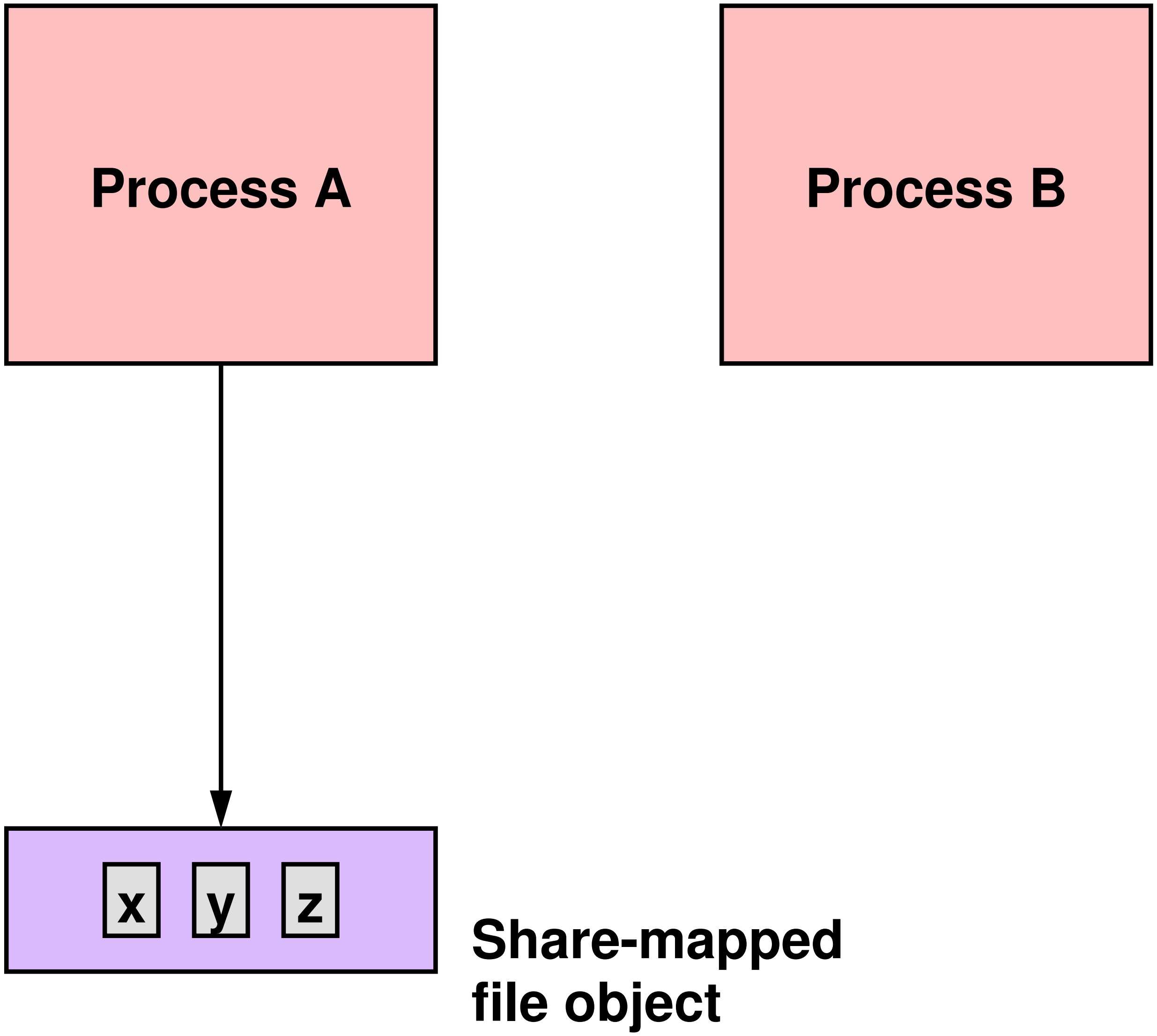
Copyright © William C. Cheng

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!

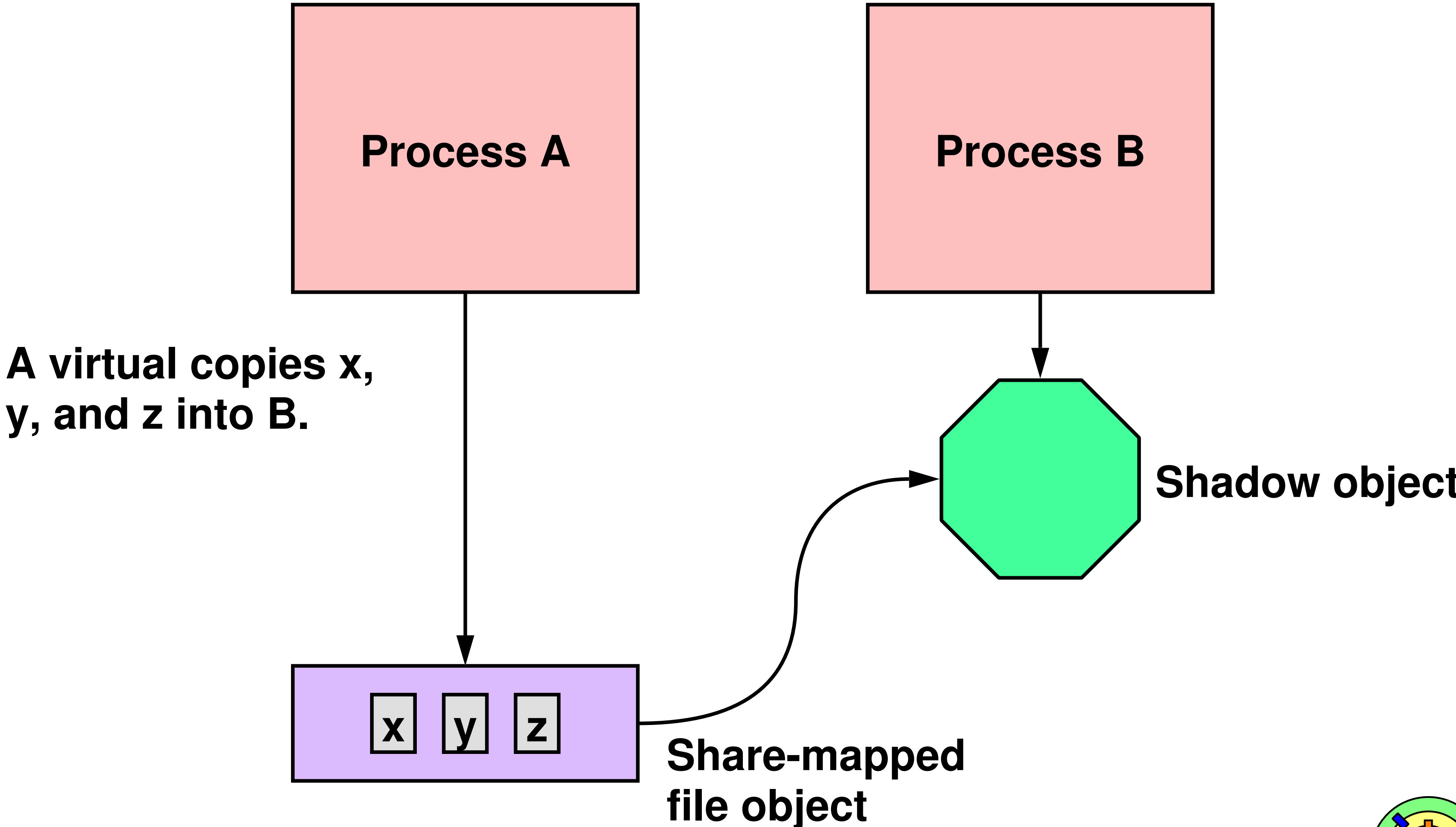
Copyright © William C. Cheng

Share and Private Mapping



Copyright © William C. Cheng

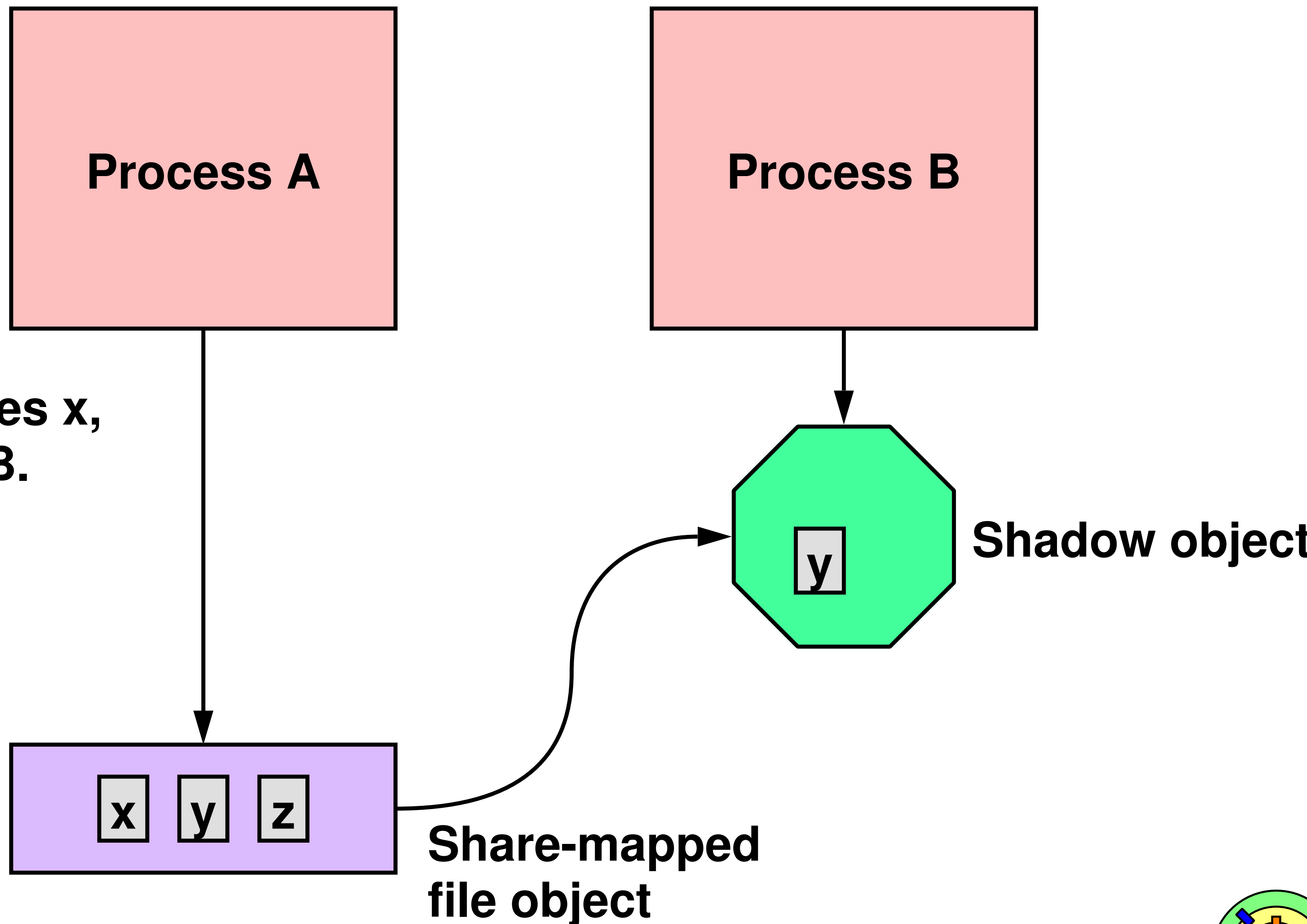
Share and Private Mapping



Copyright © William C. Cheng

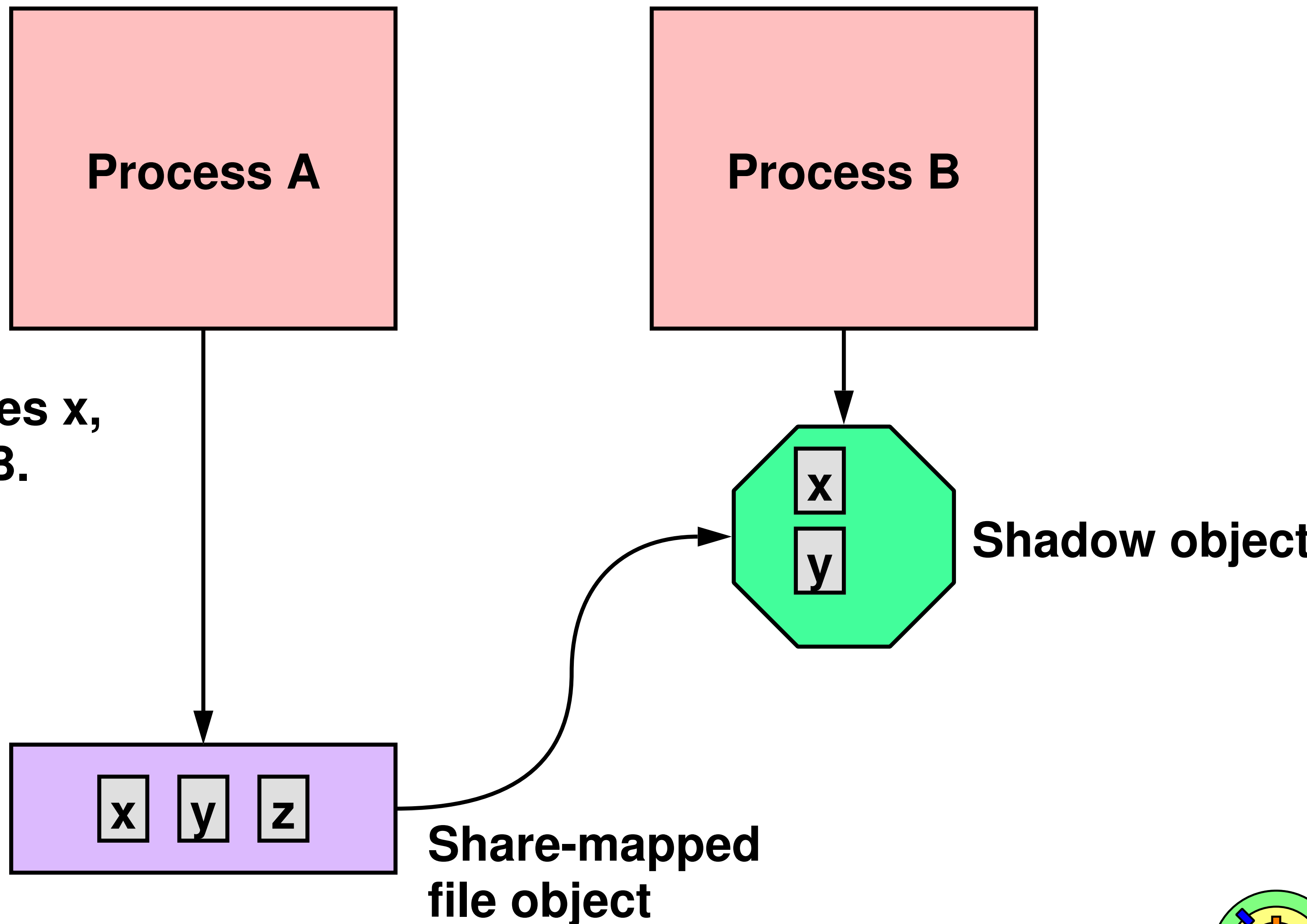
Share and Private Mapping

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



Copyright © William C. Cheng

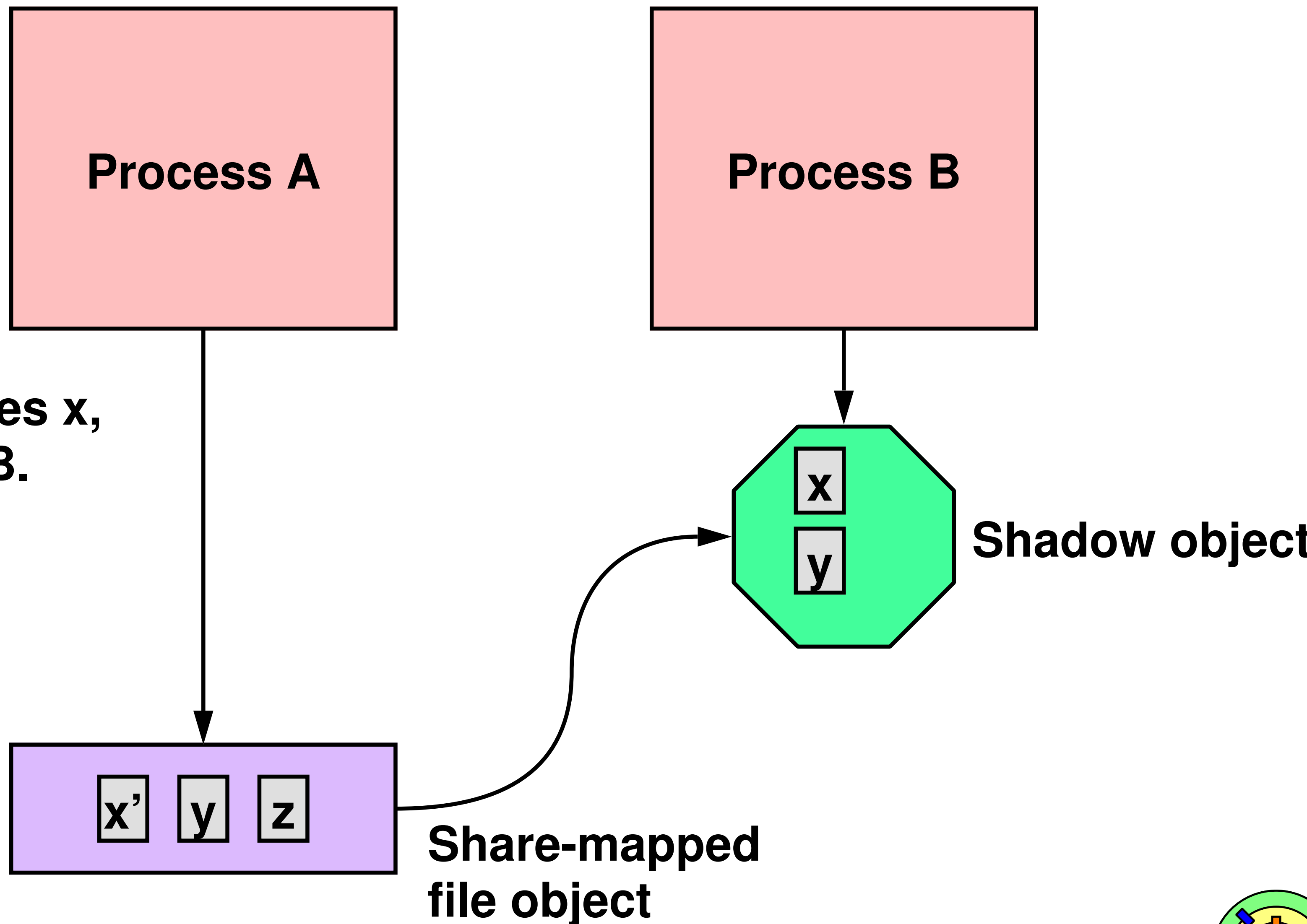
Share and Private Mapping



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

Copyright © William C. Cheng

Share and Private Mapping



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

Copyright © William C. Cheng

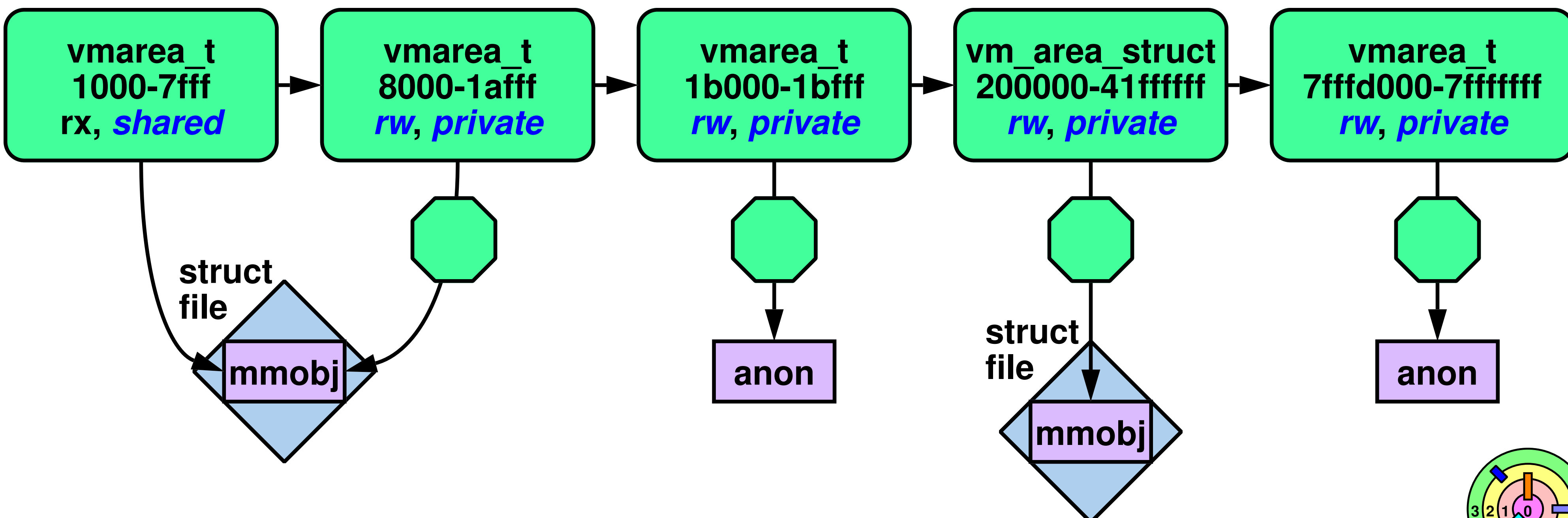
Shadow Objects Summary

- 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

Copyright © William C. Cheng

Memory Management Objects in weenix

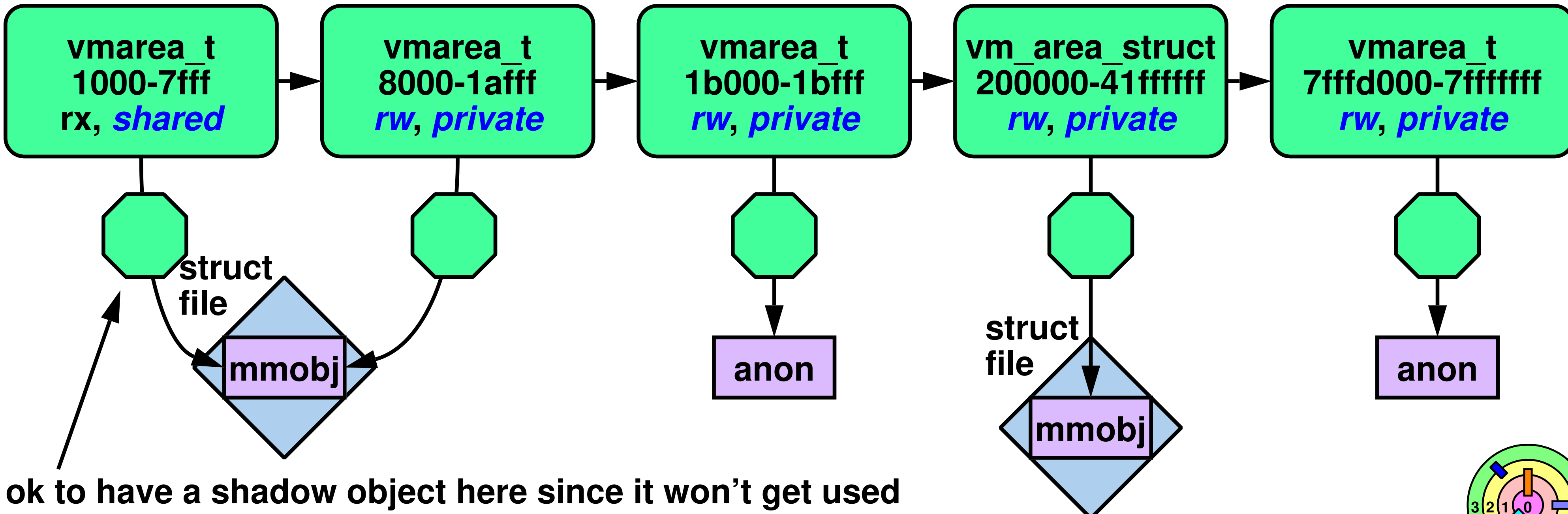
- In `weenix`, an *mmobj* is used to manage *page frames*
  - types of *mmobj* in kernel assignments are:
    - there's one that lives *inside a vnode* (`vn->vn_mmobj`)
    - a *shadow object* is an mmobj
    - an *anonymous object* (meaning not associated with a file and not a shadow object) is an mmobj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



Copyright © William C. Cheng

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

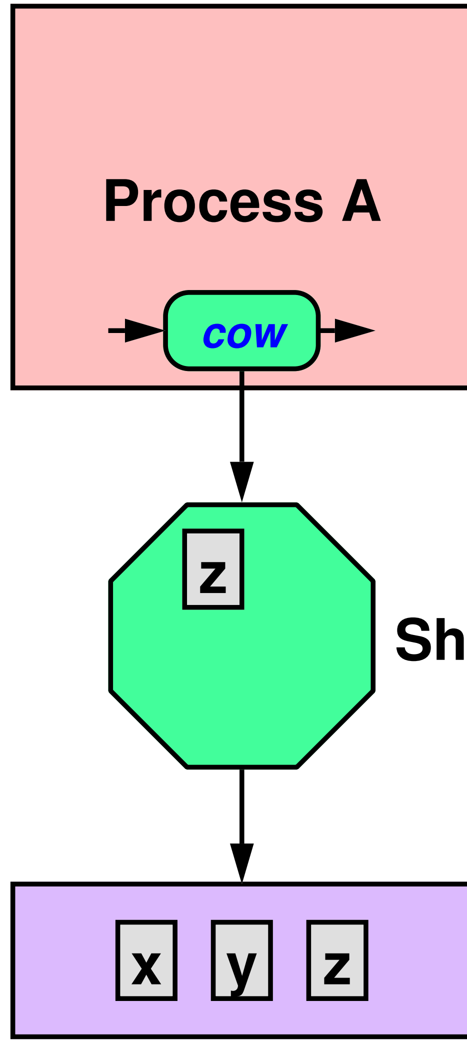


ok to have a shadow object here since it won't get used since it's read-only (i.e., no copy-on-write is possible)

Copyright © William C. Cheng

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)`

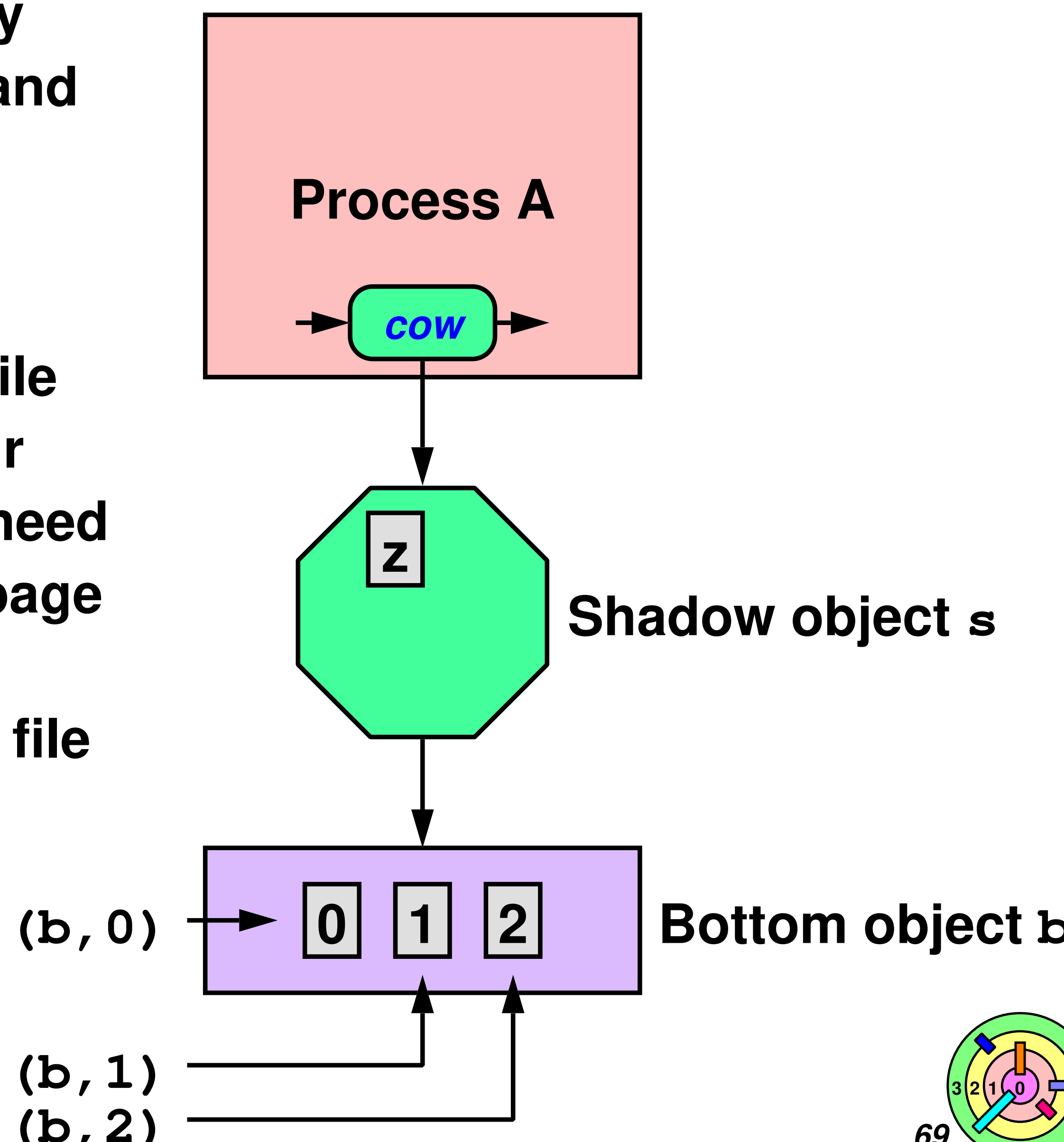


where does `pagenum` come from?
- think of a file as an array of pages
- then `pagenum` is the *array index* into the file

Copyright © William C. Cheng

Memory Management Objects in `weenix`

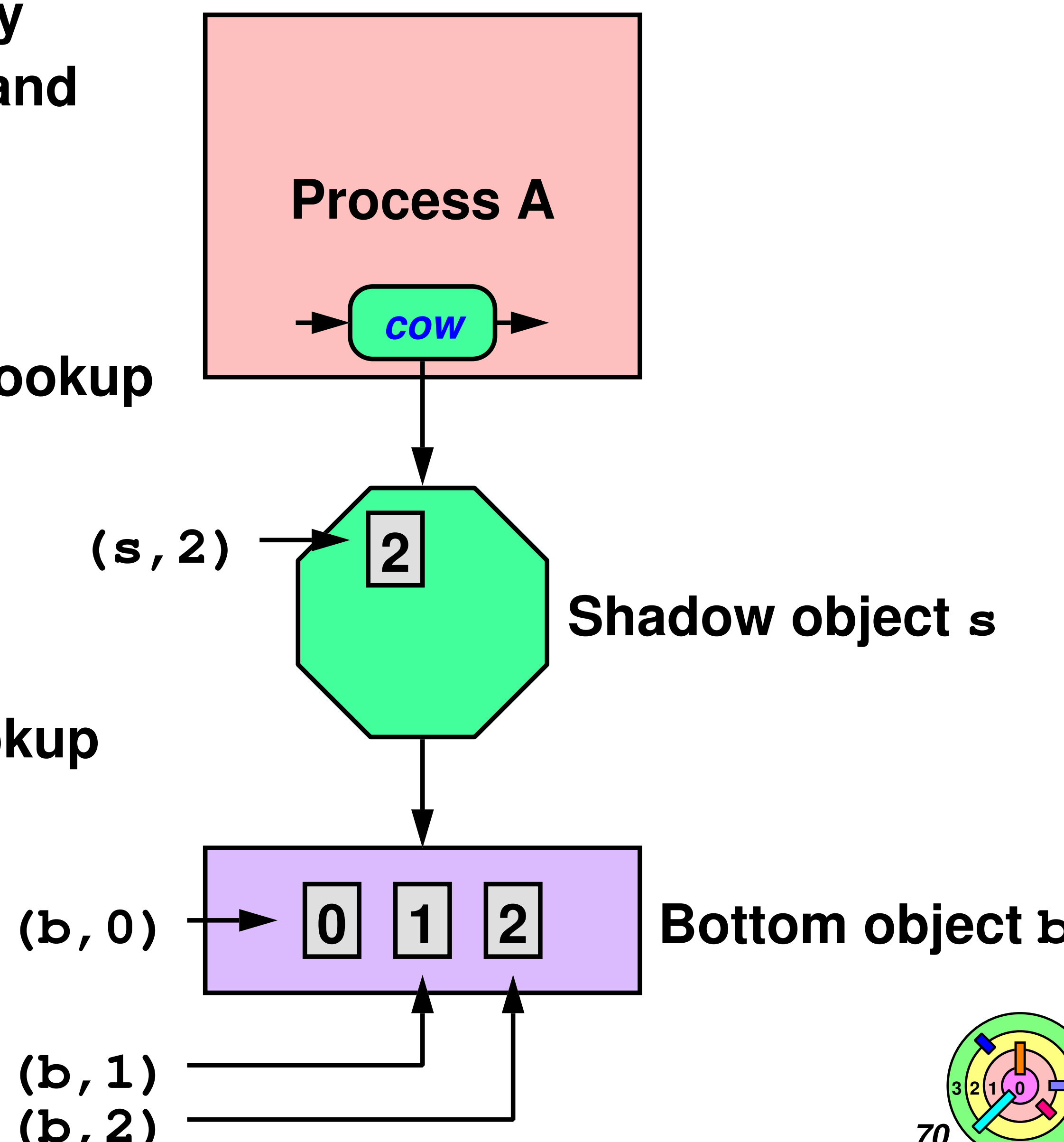
- In `weenix`, an *mmobj* is used to manage *page frames*
  - a page frame is uniquely identified by an `mmobj` and a `pagenum`
    - notation: `(mmobj, pagenum)`
    - if you map part of a file (say a page) into your address space, you need to remember which page
      - `pagenum` is then a *page index*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



Copyright © William C. Cheng

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*



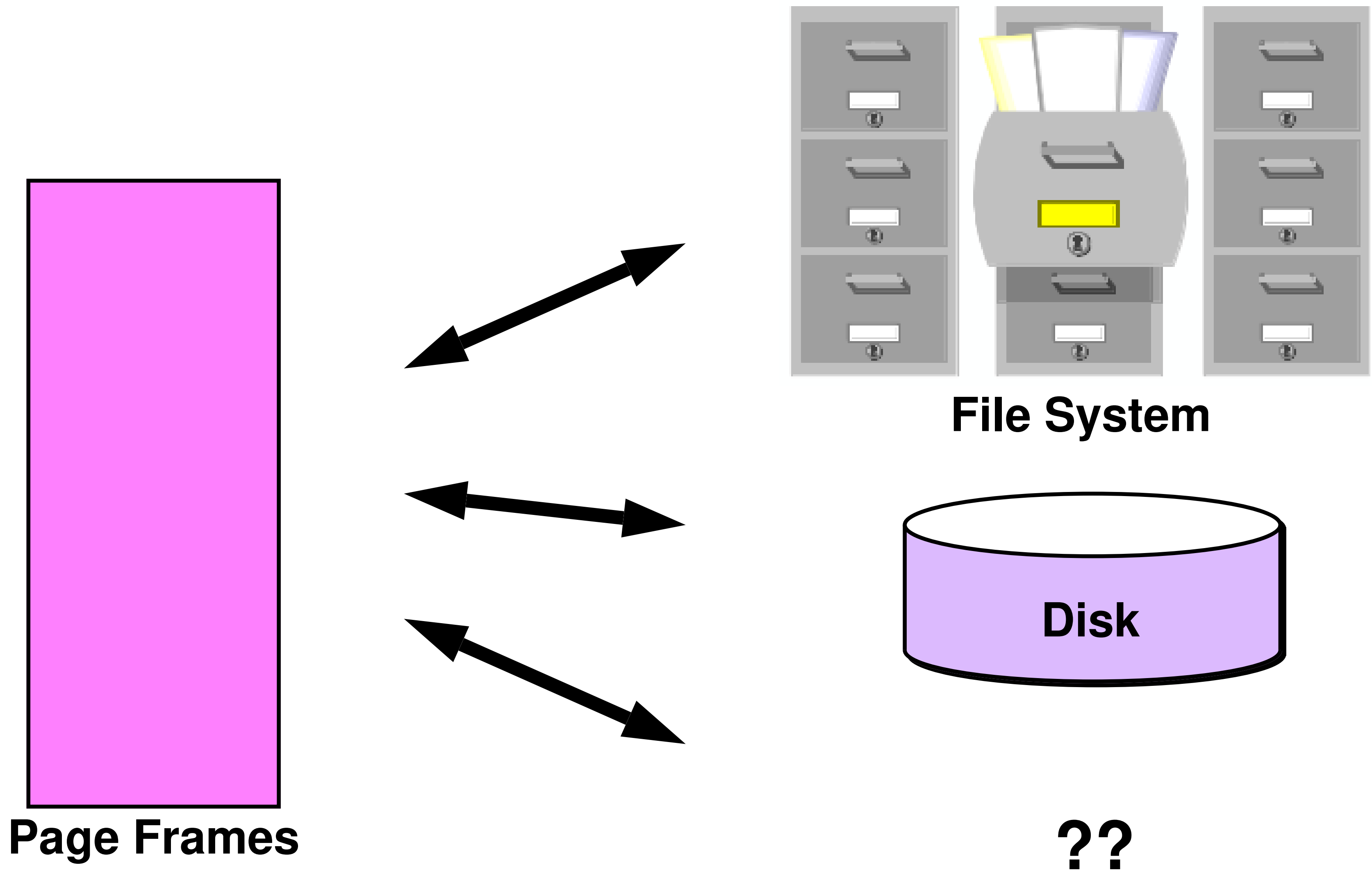
Copyright © William C. Cheng

7.3 Operating System Issues

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

Copyright © William C. Cheng

The Backing Store

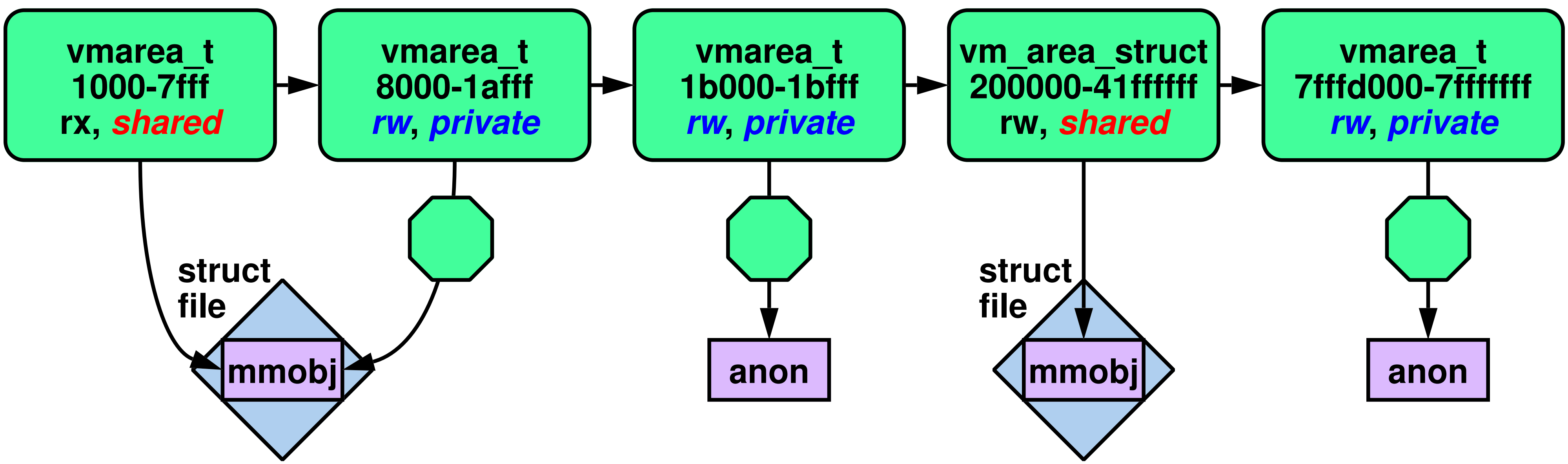


Copyright © William C. Cheng

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"

Copyright © William C. Cheng

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()`

Copyright © William C. Cheng

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

Copyright © William C. Cheng

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

Copyright © William C. Cheng

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

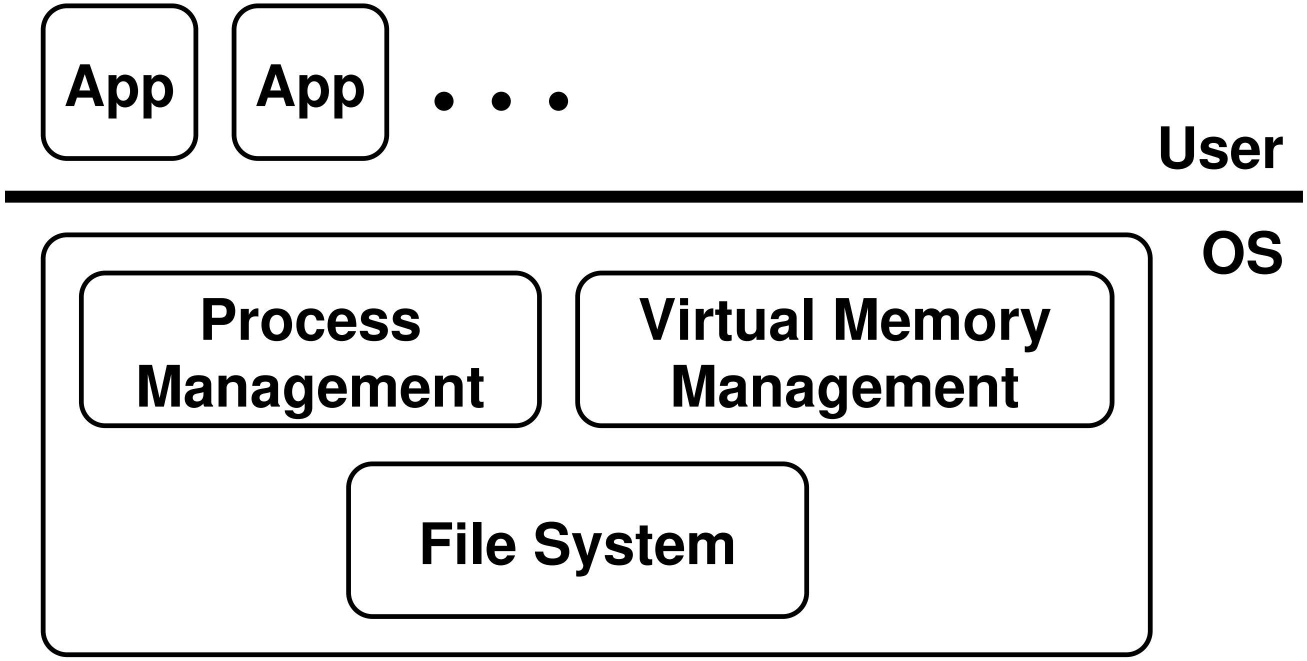
Copyright © William C. Cheng

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

Copyright © William C. Cheng

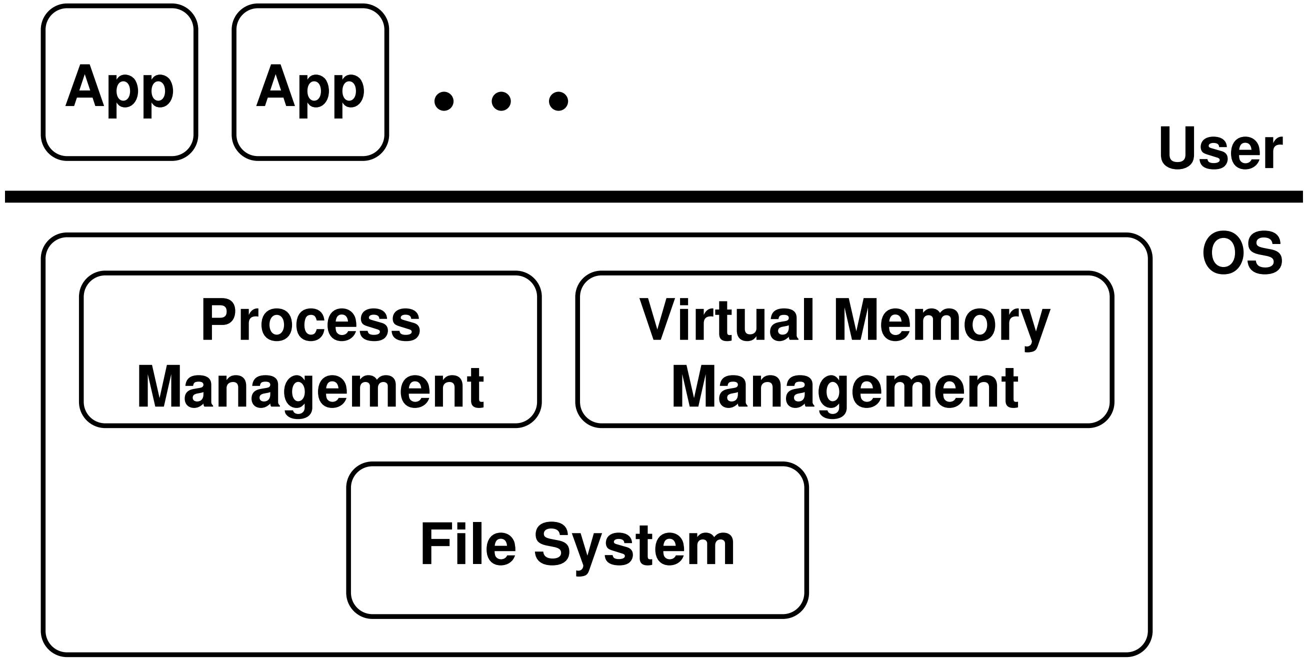
Summary



- 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

Copyright © William C. Cheng

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()`)

Copyright © William C. Cheng