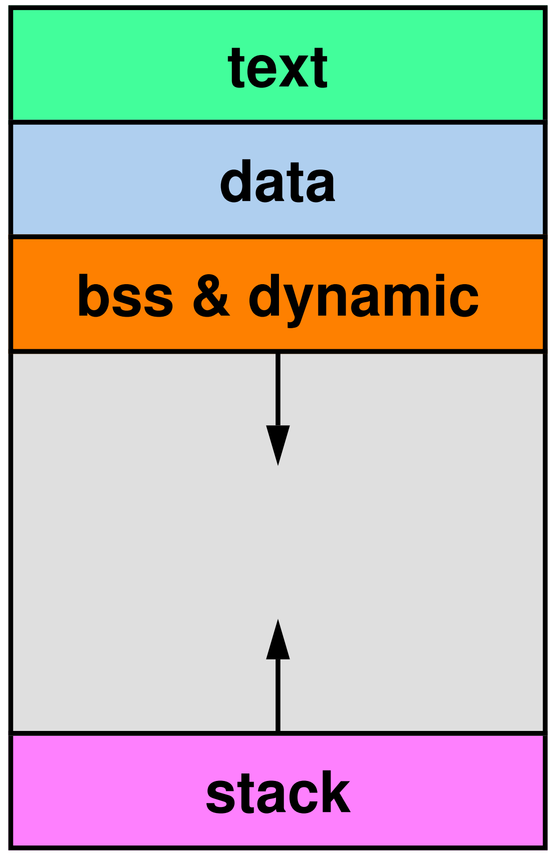
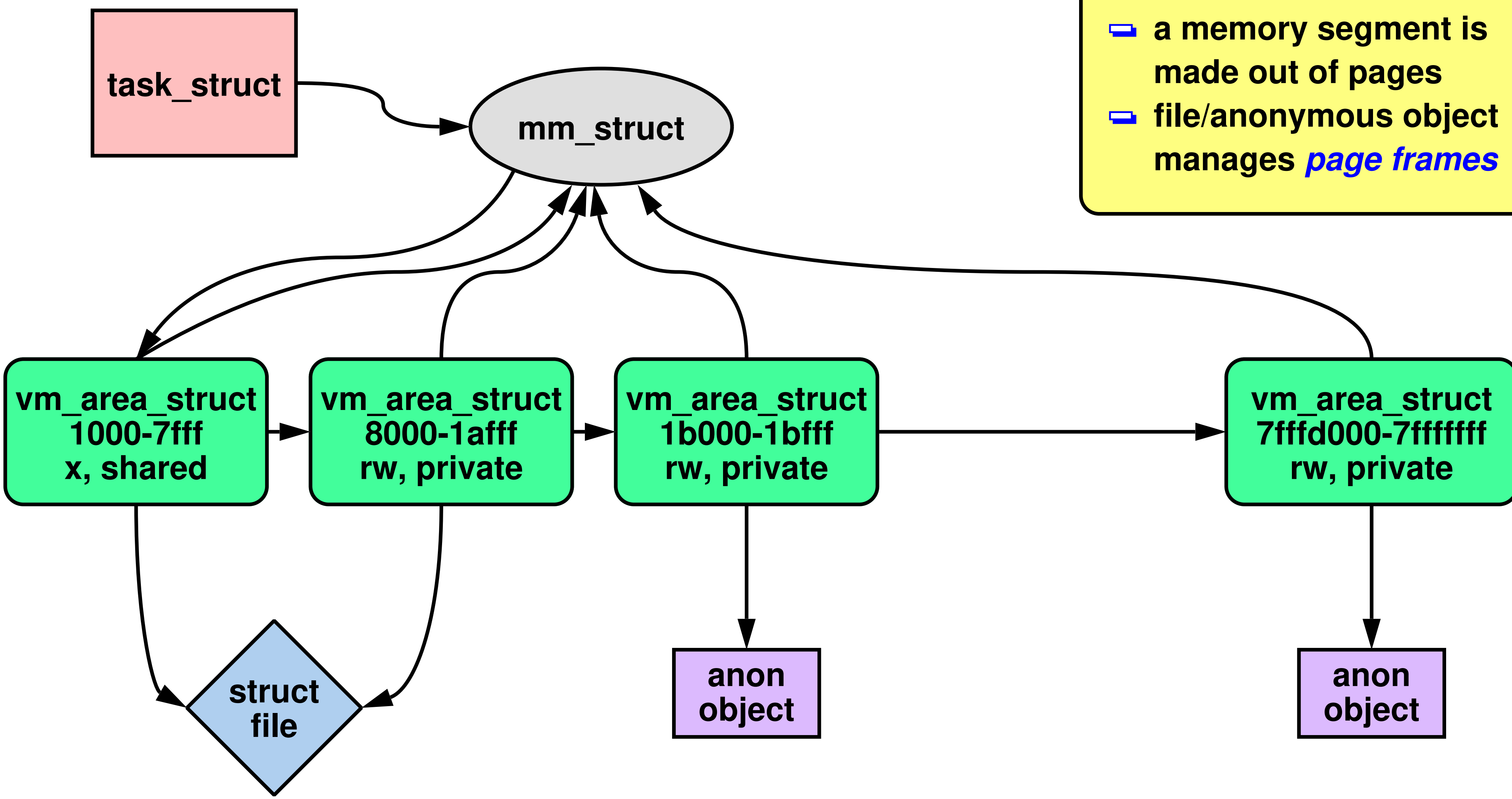
Simple User Address Space



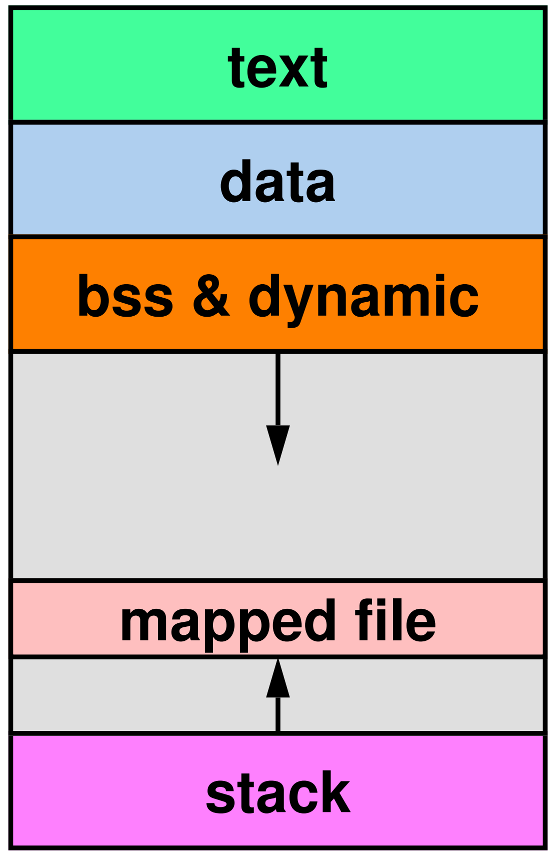
Address-Space Representation (Somewhat Simplified)



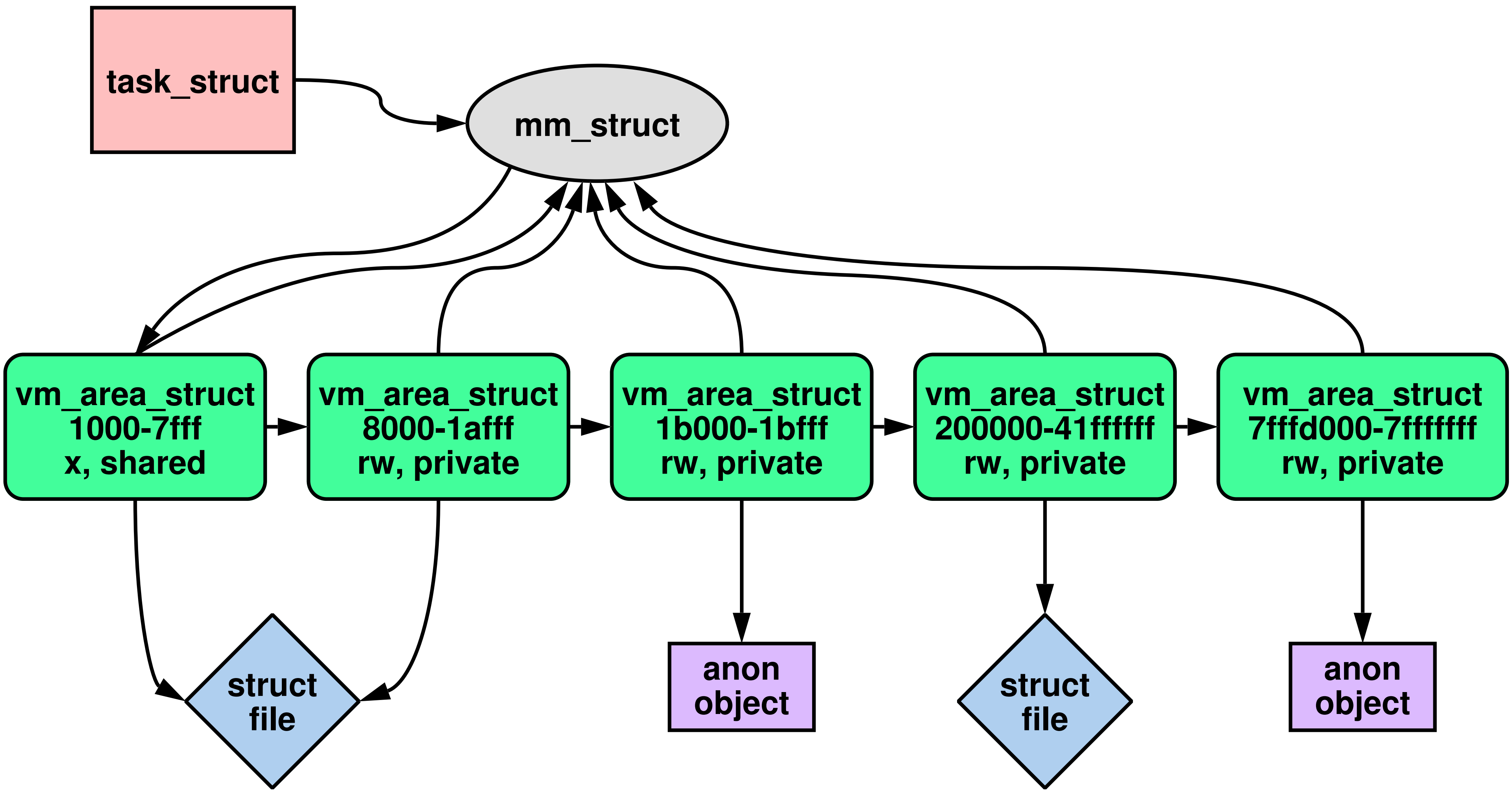
- a memory segment is made out of pages
- file/anonymous object manages *page frames*

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

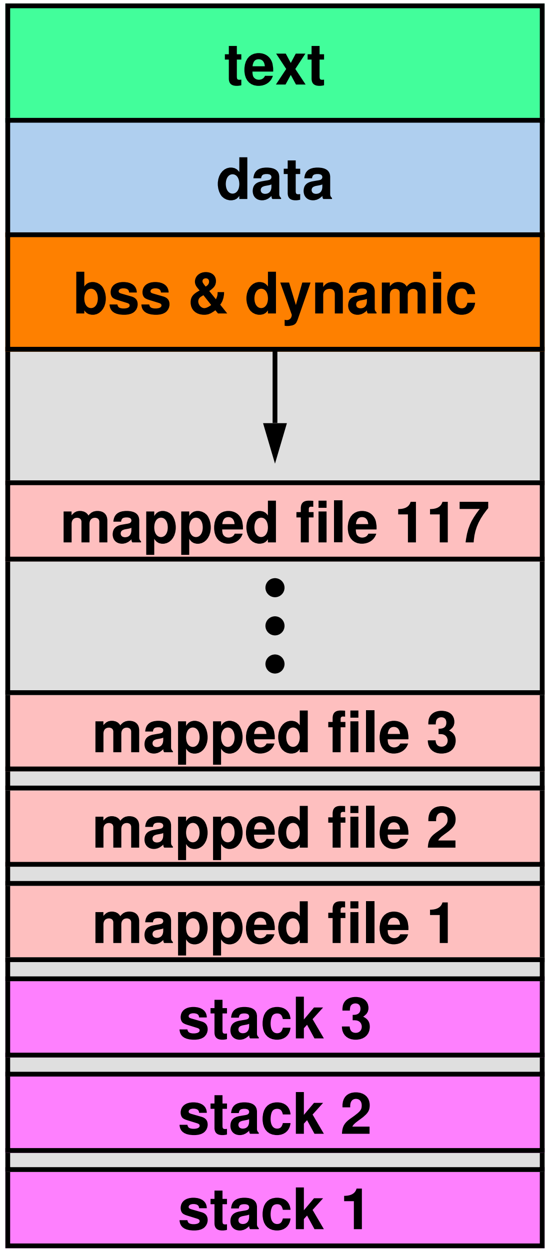
Adding a Mapped File



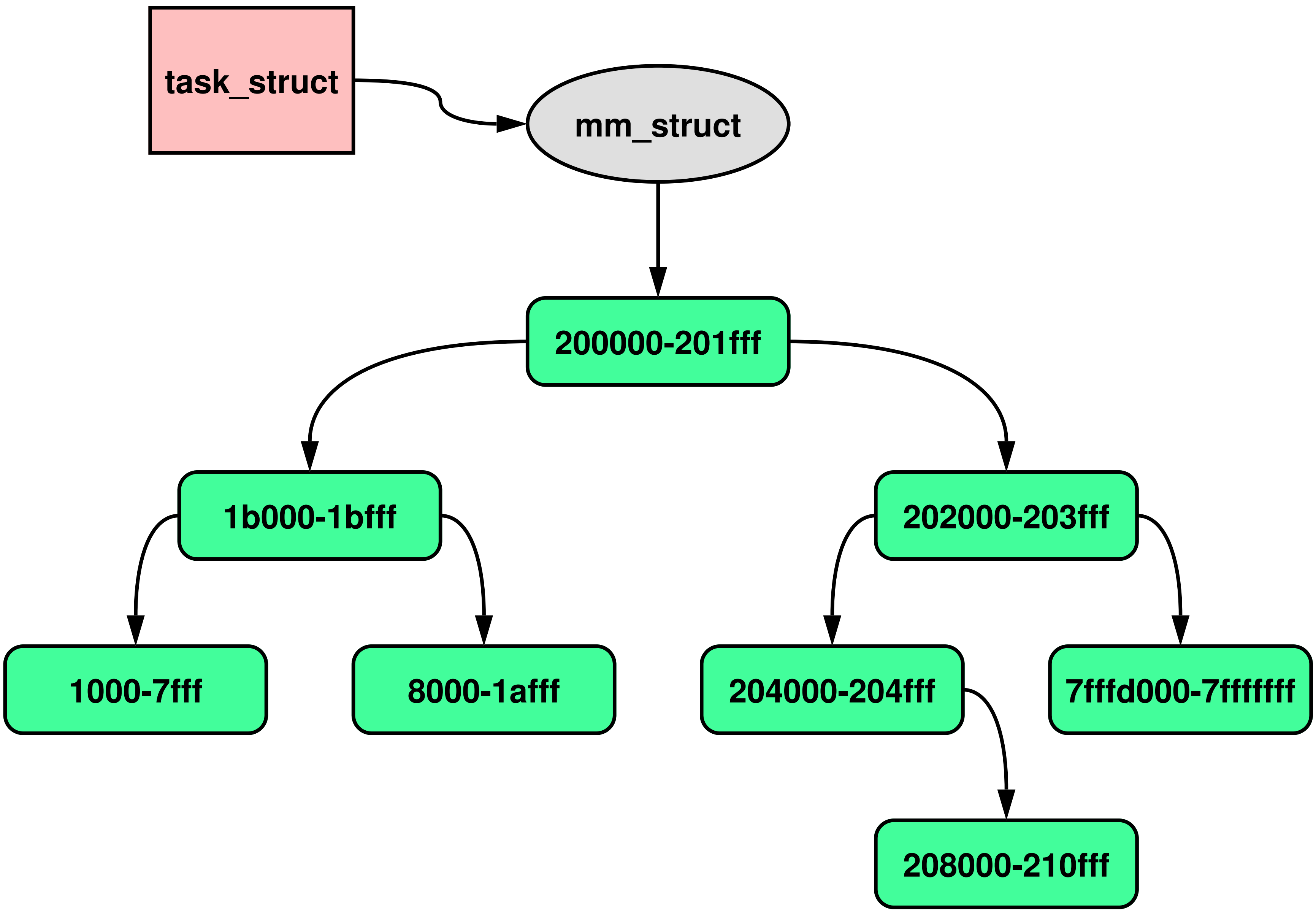
Address-Space Representation: More Areas



Adding More Stuff



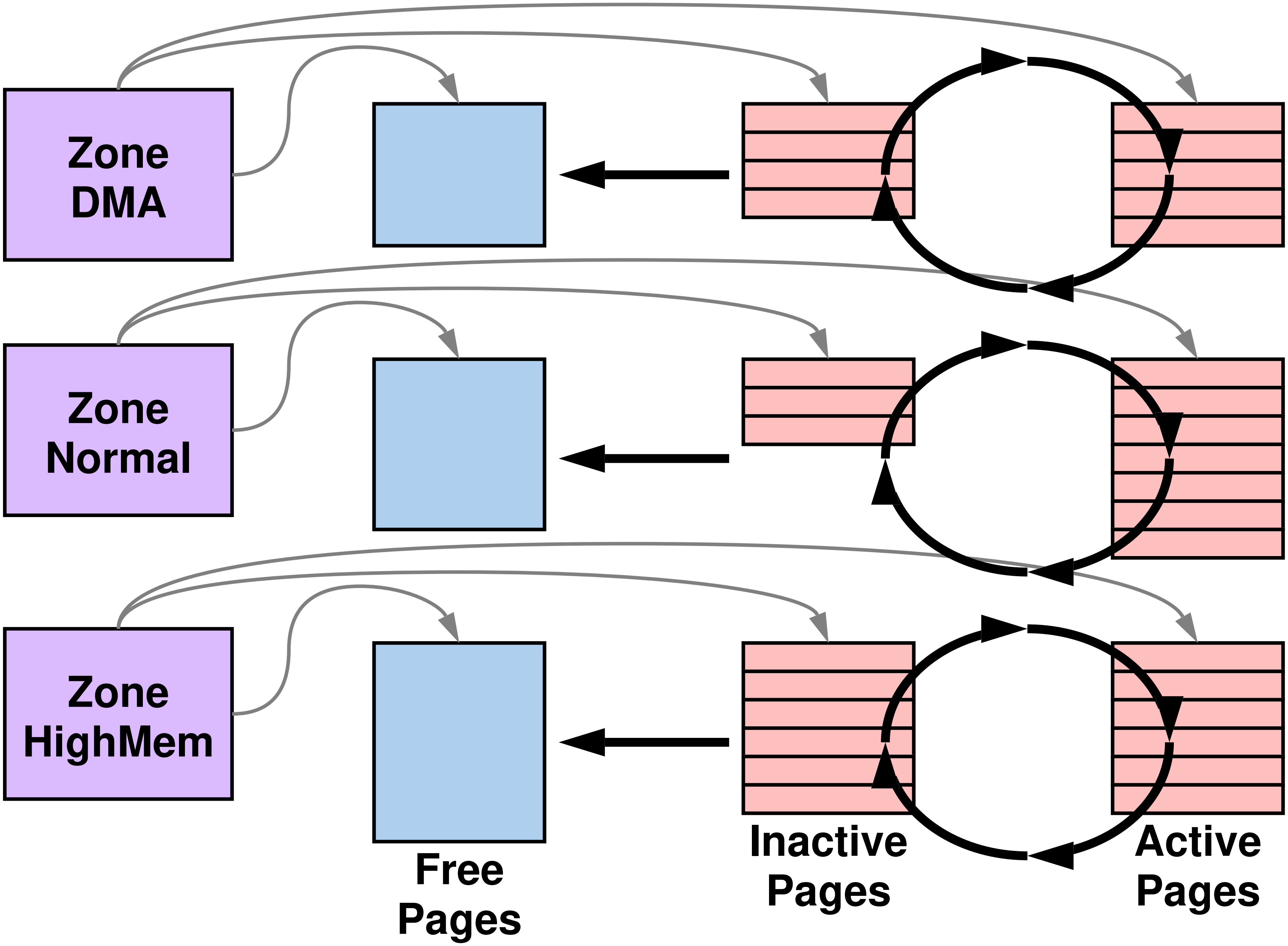
Address-Space Representation: Reality



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

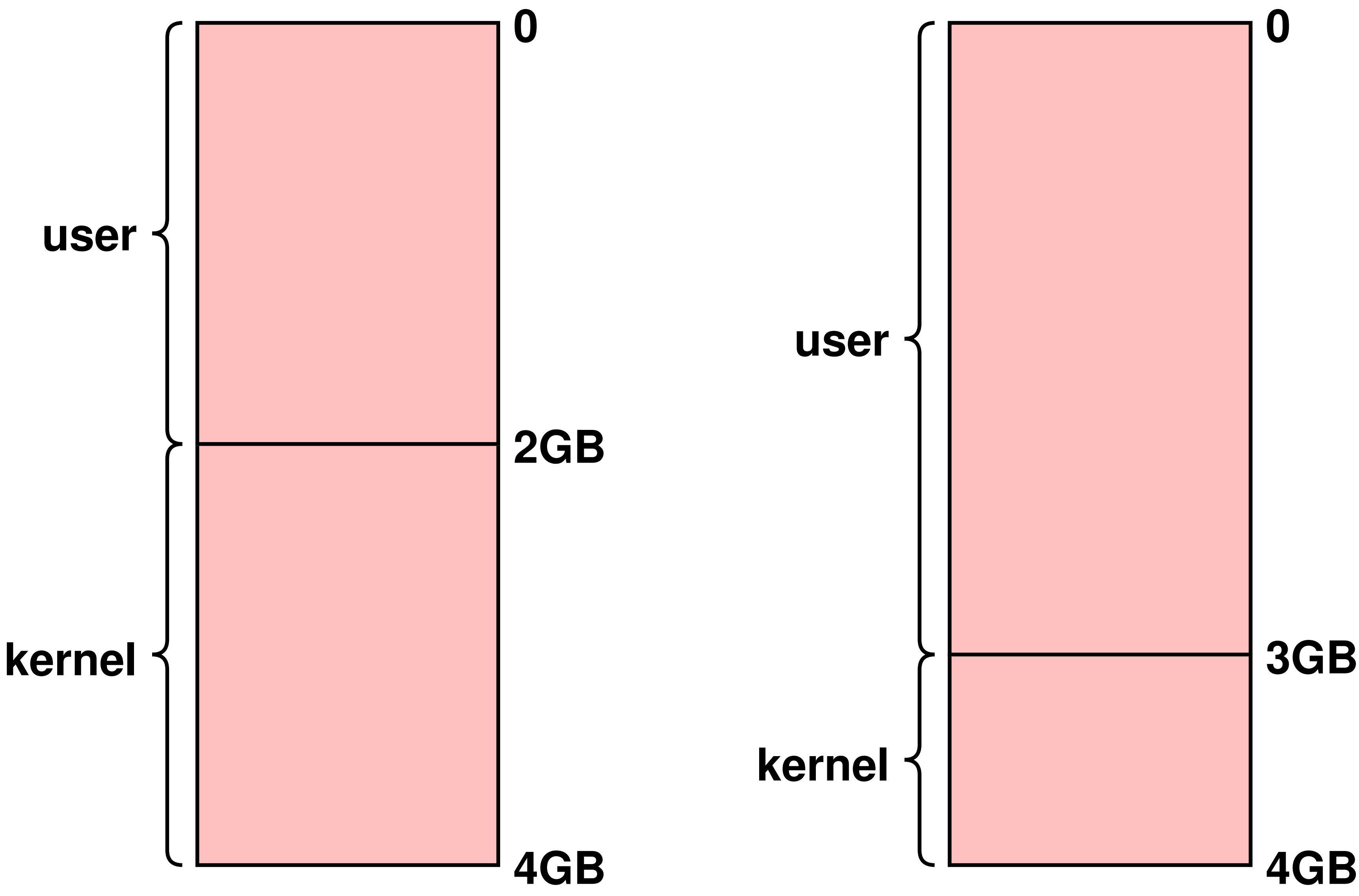
- For each process, *PCB* contains
  - *virtual memory map* (which represents the user address space)
    - maps *virtual* memory segments
    - keeps track of *page frames* and *backing store*
  - hardware page tables
- Globally, *free, active, and inactive page list* are maintained
- Example usage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

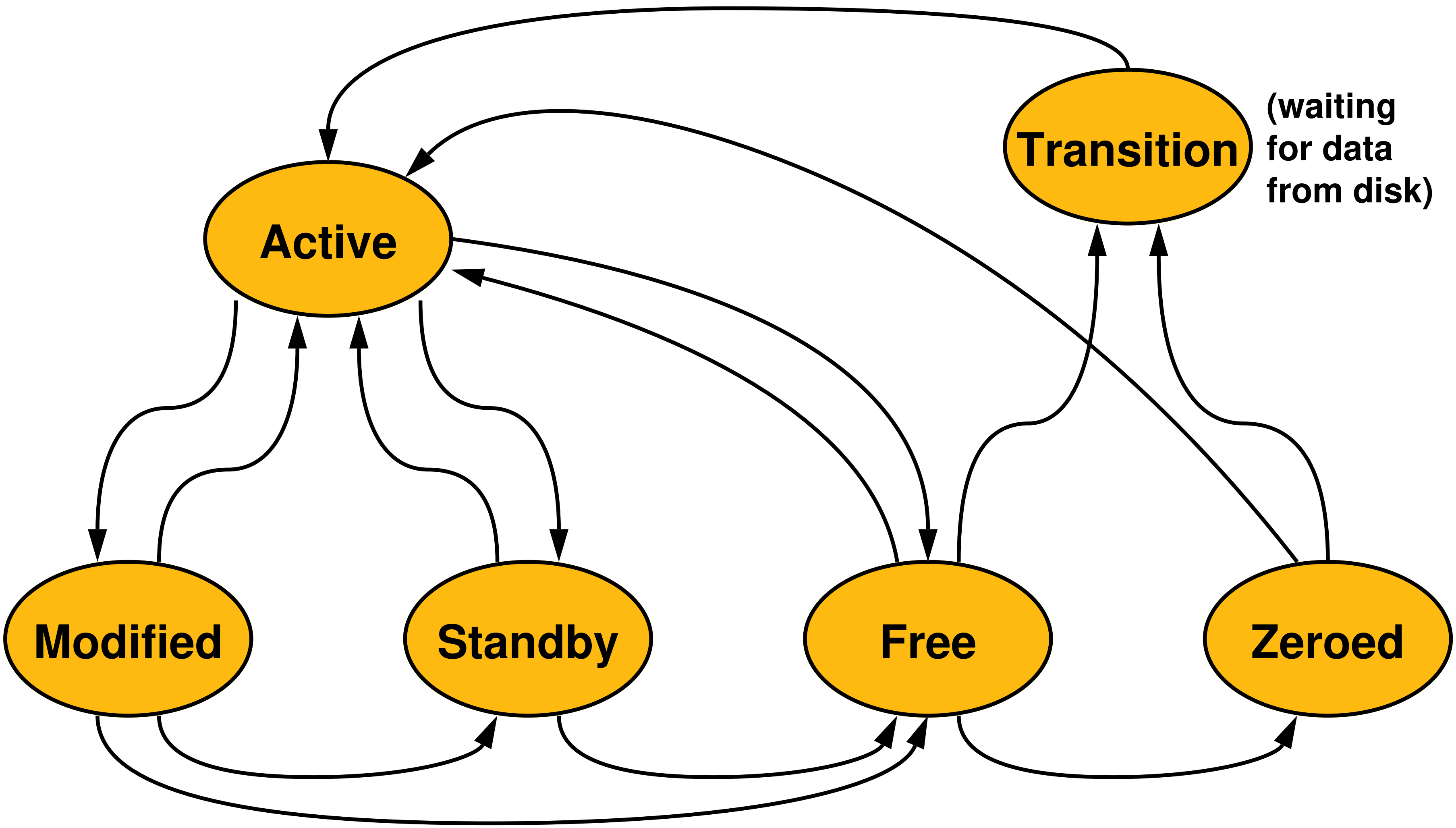
- Two choices



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

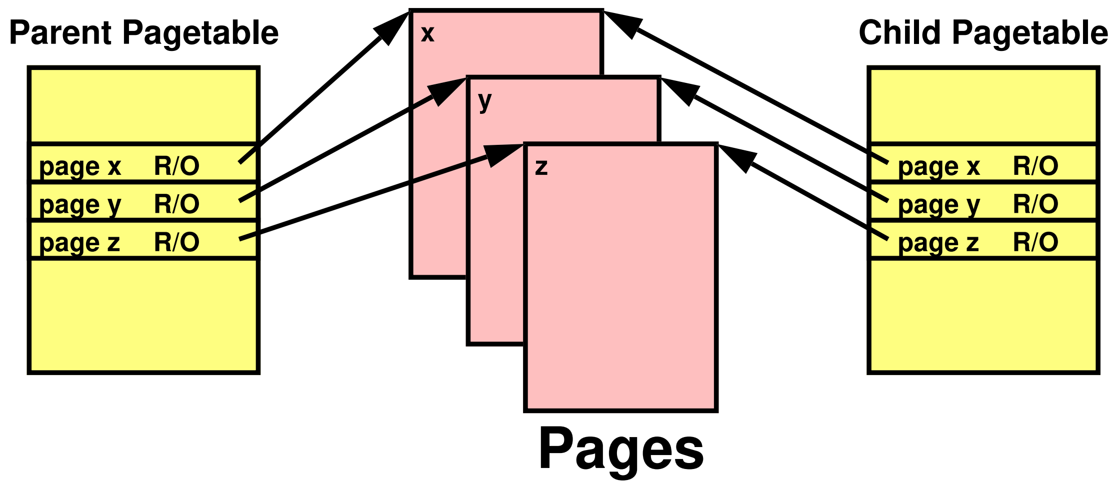
A Better `fork()`

- 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()`

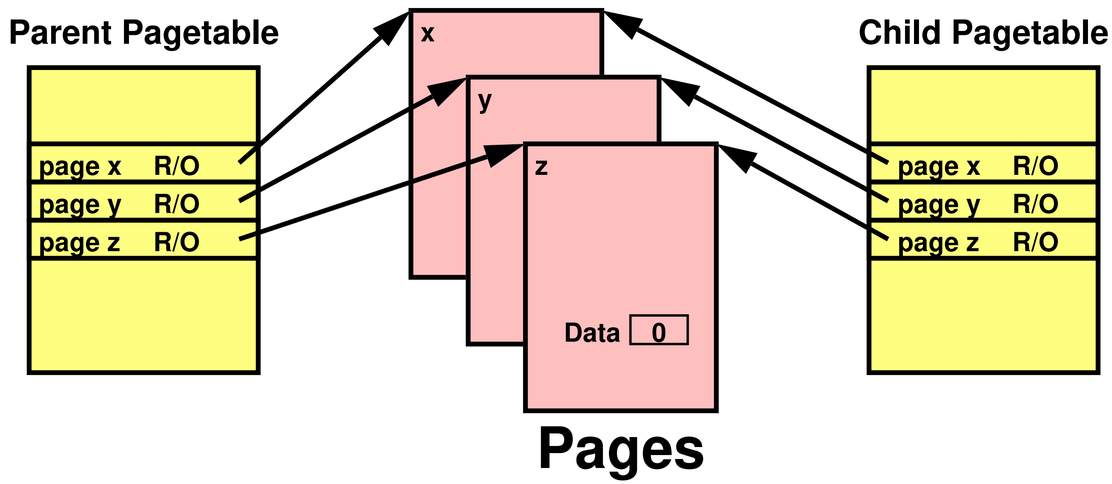
- To implement the "better" `fork()`, we need to use *copy-on-write*
  - a process gets a *private* copy of a page after a thread in the process performs a *write* to that page for the *first time*
    - set every PTE to *R/O* for pages that correspond to memory segments that needs *copy-on-write* (i.e., 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

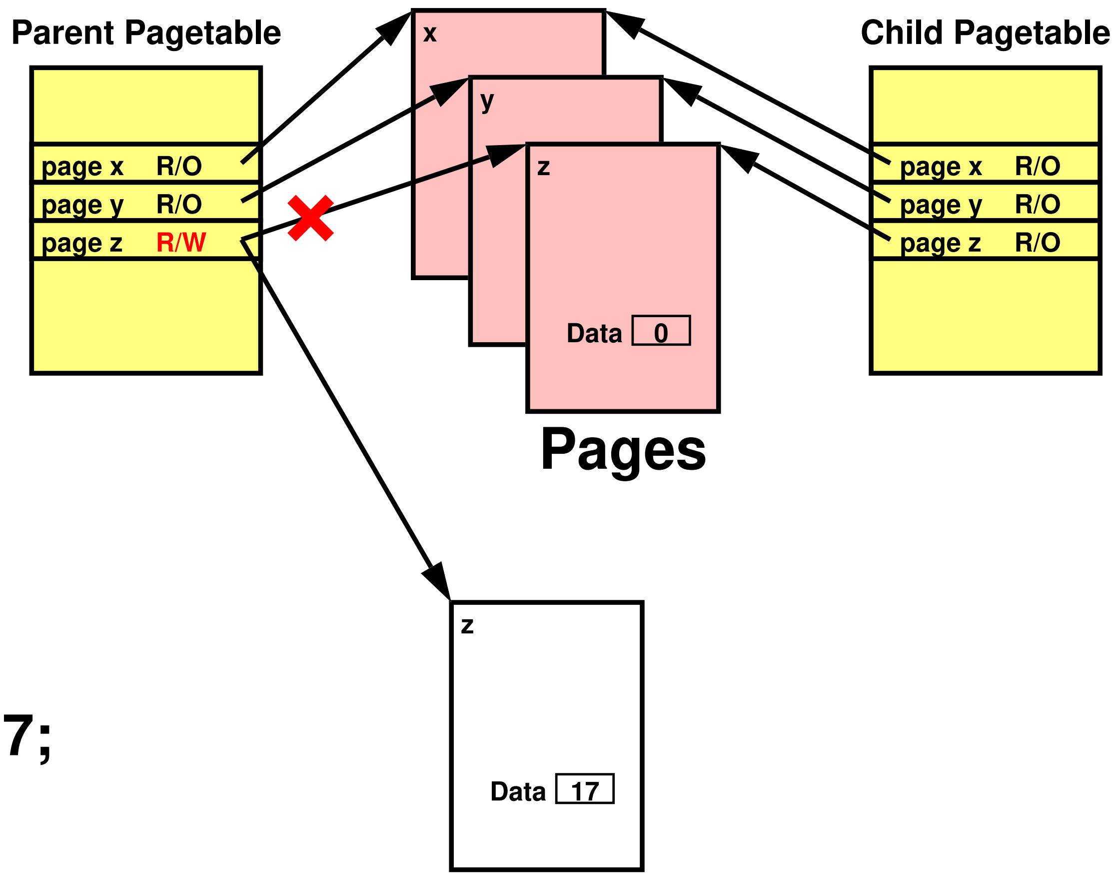
Private Mapping - Copy on Write Occurs after `fork()`



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*

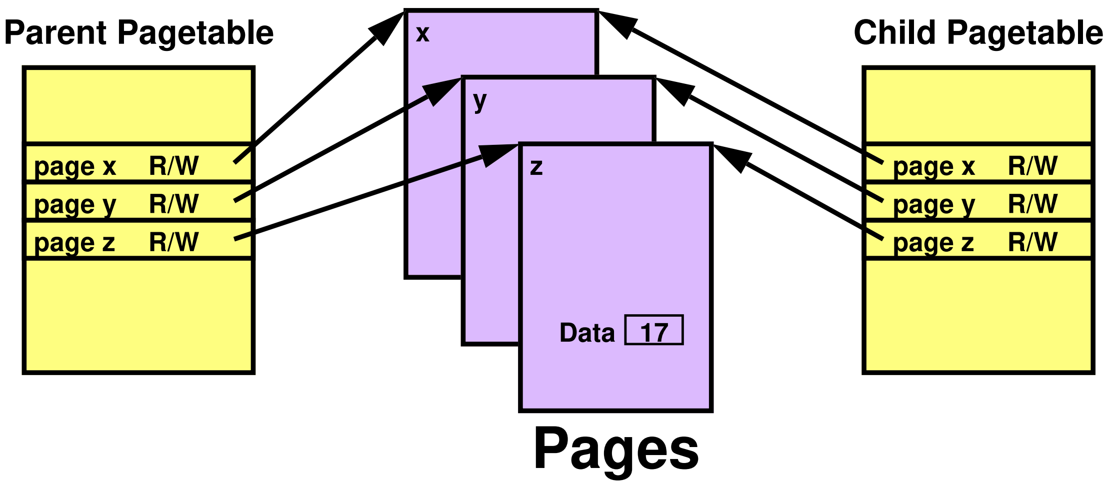
Private Mapping - Copy on Write Occurs after `fork()`



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***

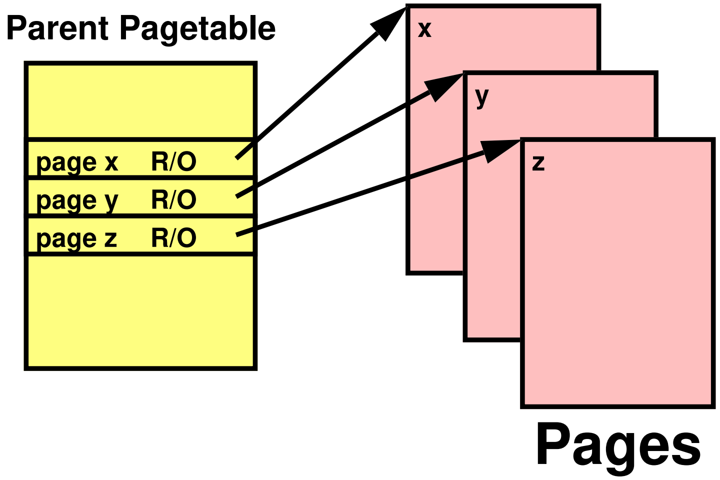
Share-Mapped Files



Data = 17;

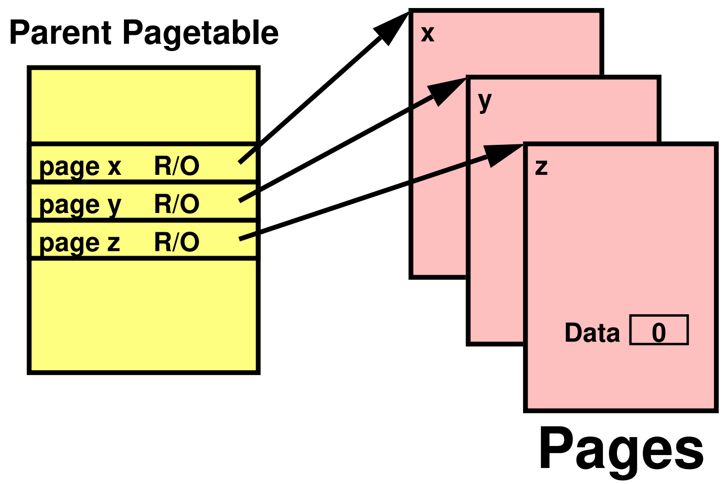
- For *shared* mapping, changes are writting into the shared page
  - please note that the information about whether a page is *shared* or *private* is *not* inside the page table
    - it is kept in a kernel data structure (`vm_area_struct`)

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



For *private* mapping, *copy on write*

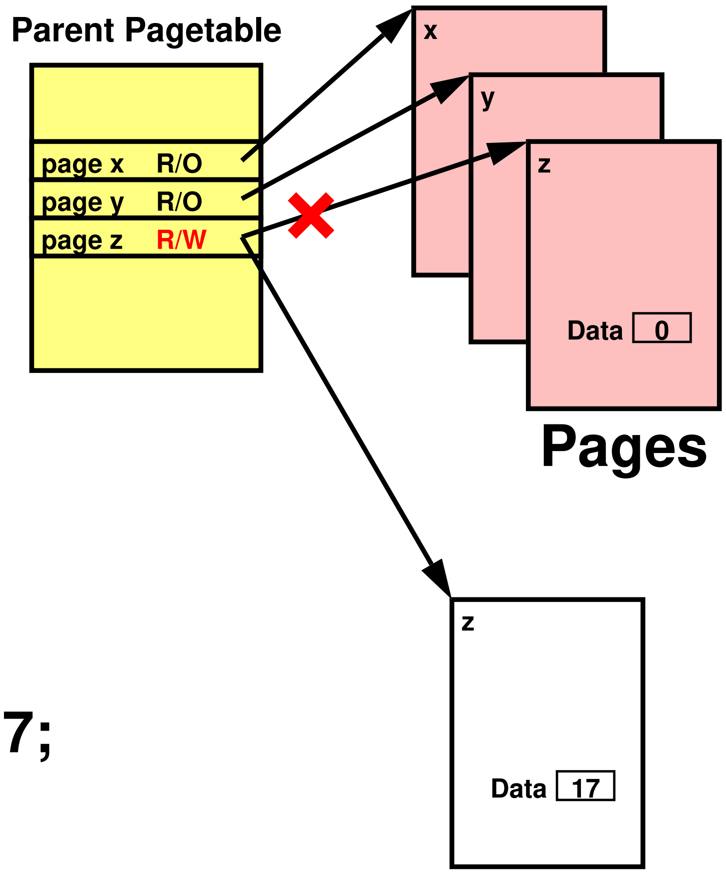
Private Mapping - Copy on Write before `fork()`



Data = 17;

For *private* mapping, *copy on write*

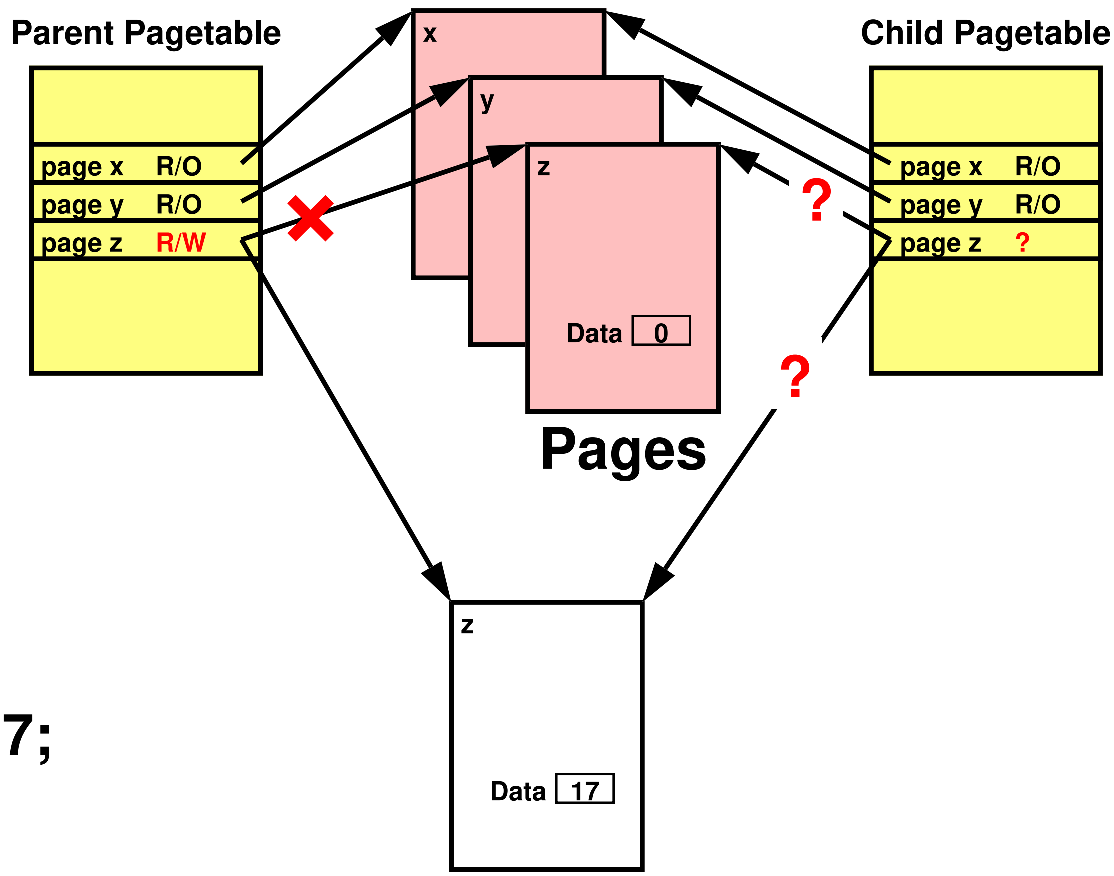
Private Mapping - Copy on Write before `fork()`



Data = 17;

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

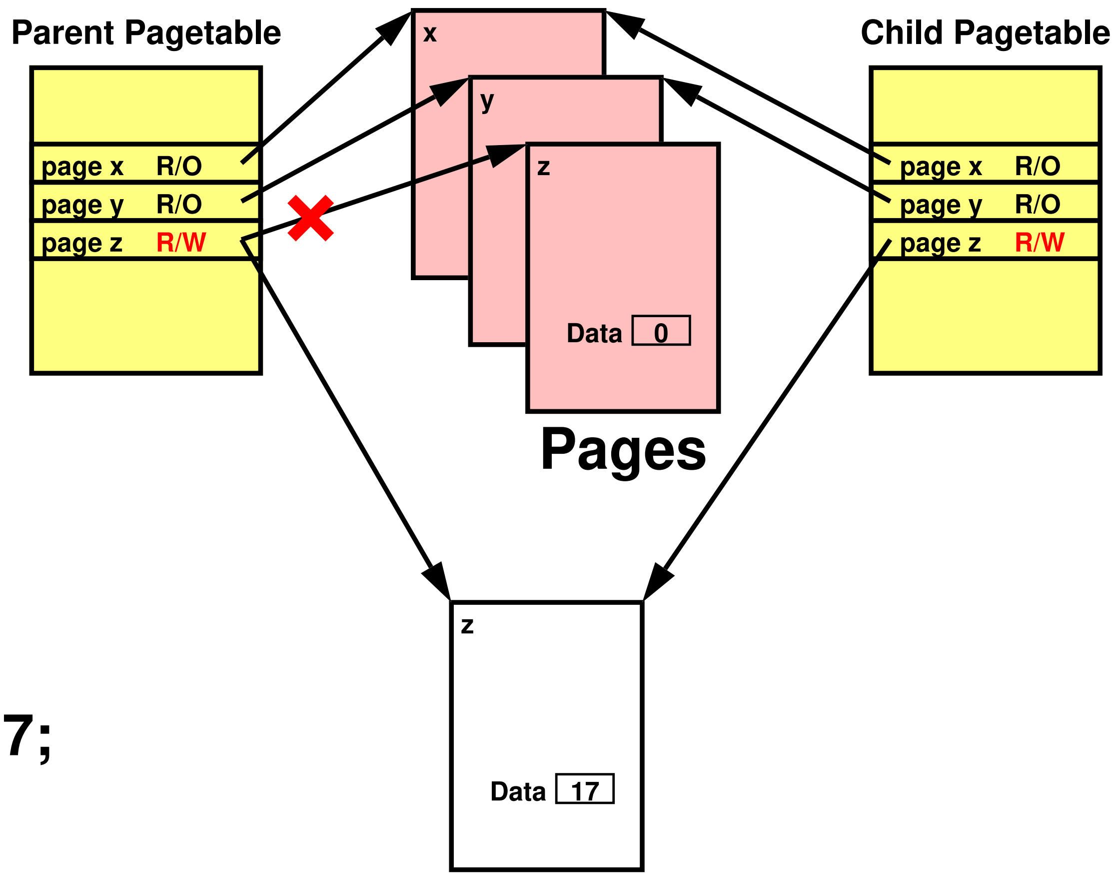
A Private-Mapped File Changes



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**

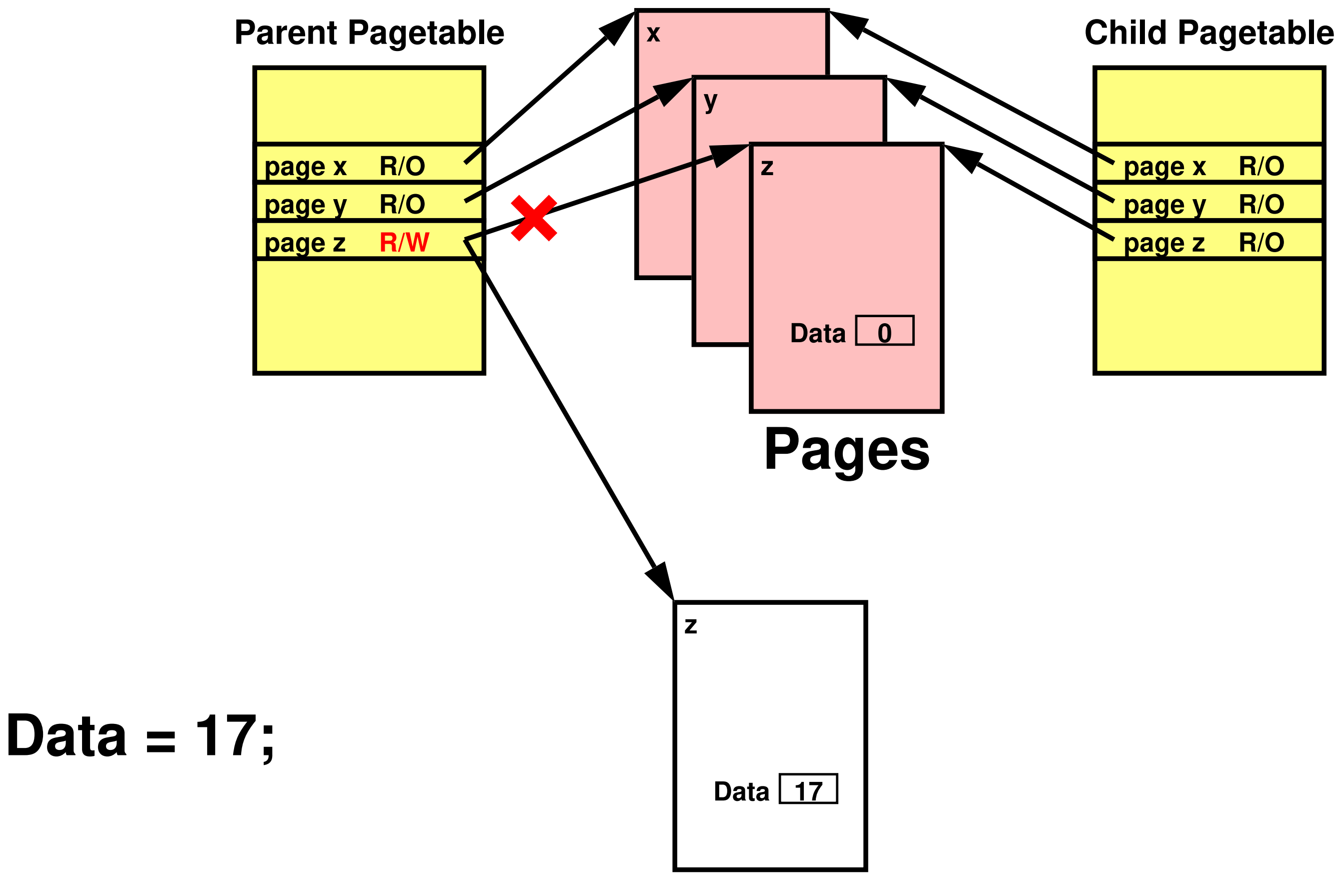
A Private-Mapped File Changes



Data = 17;

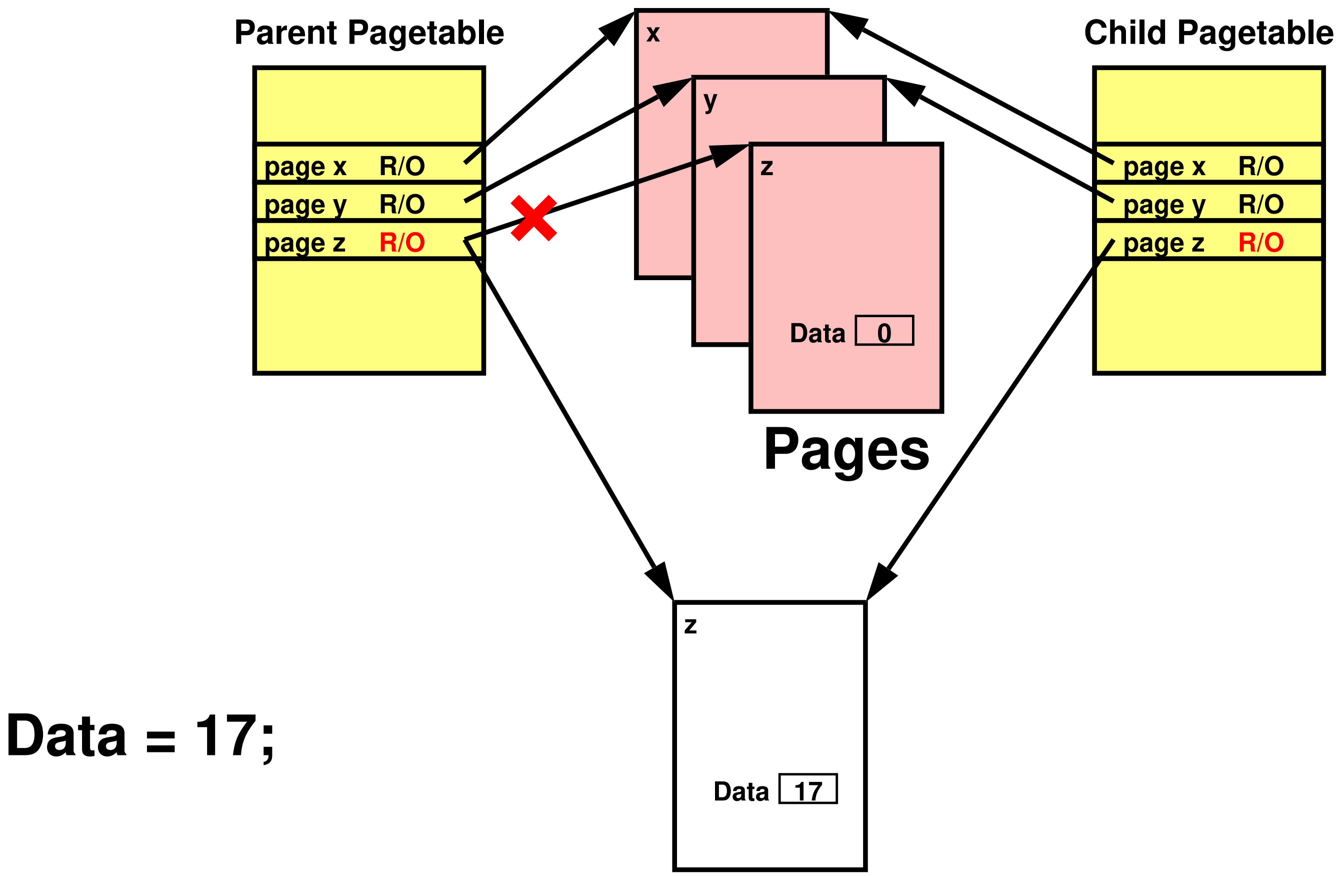
- Complication: what if the page is modified before `fork()`?
  - `memcpy()` the parent's page table is wrong: what if the parent modify the page further?
    - child should not see these changes

A Private-Mapped File Changes



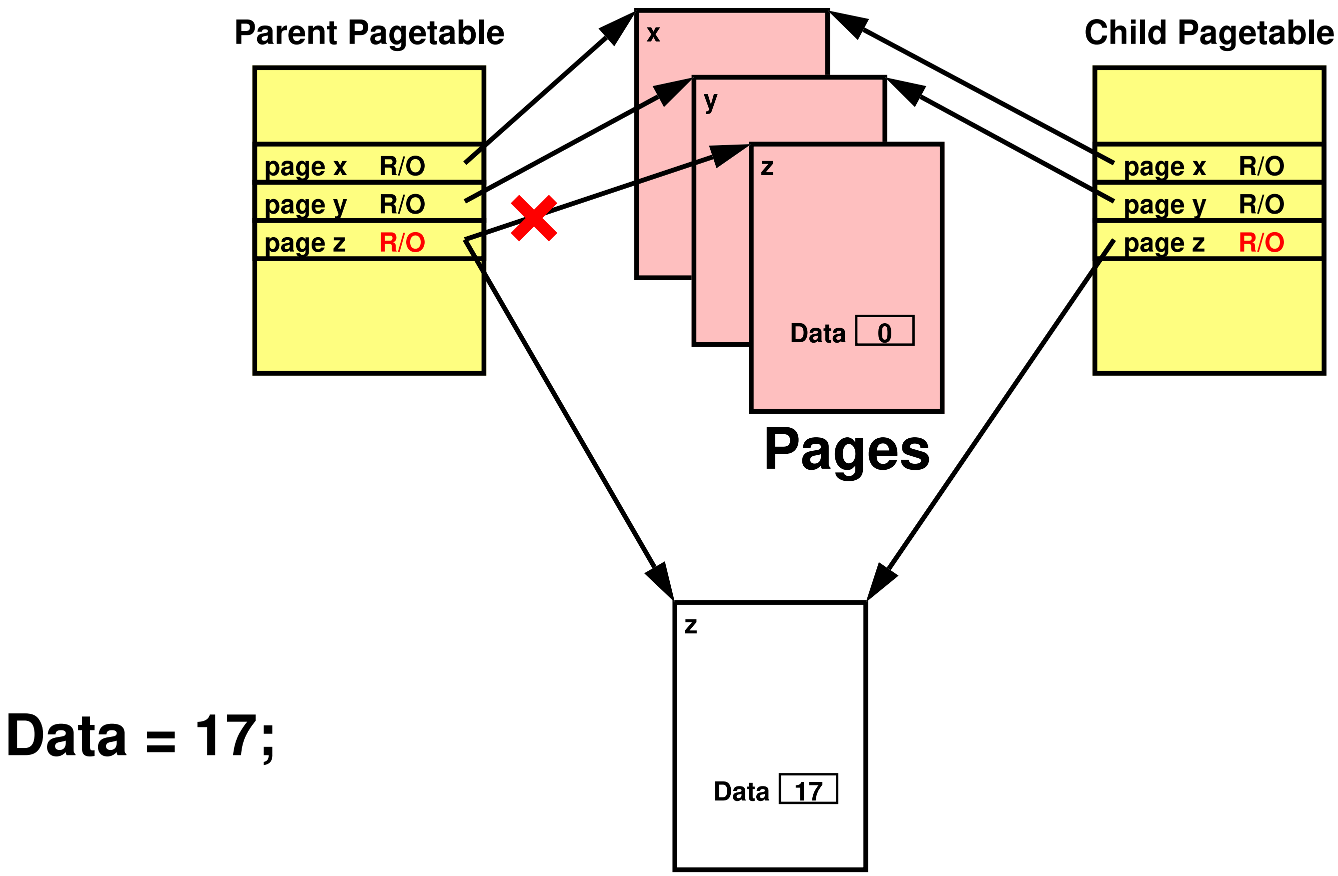
- Complication: what if the page is modified before `fork()`?
  - this is also wrong
    - child process should see 17 in Data on page z

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



- 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)