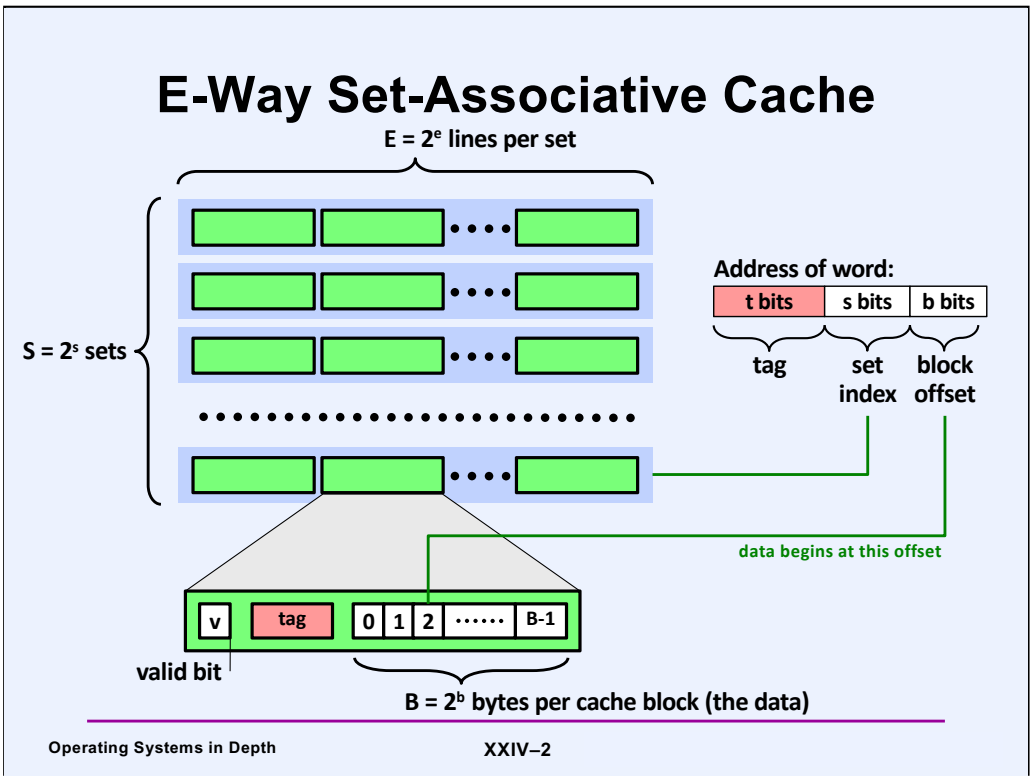
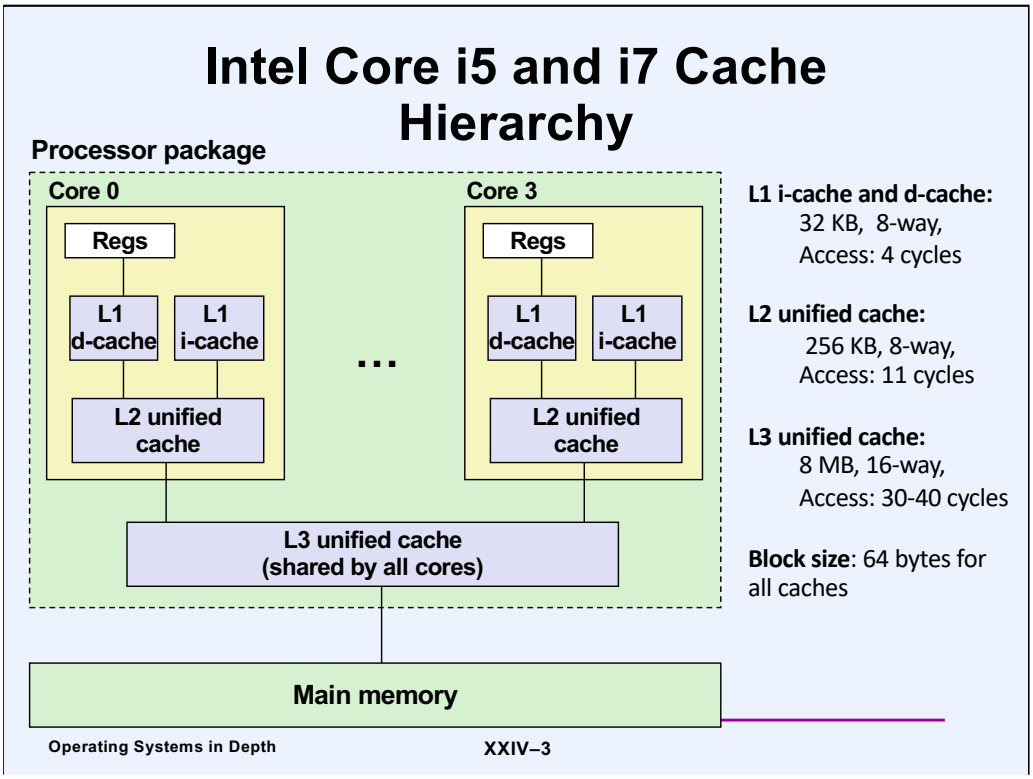
Memory Management Part 5



This slide, from CMU and used in CS 33, shows the organization of an E-way set associative cache. All blocks whose addresses have the same set index will map to the same cache set, but only E such blocks will fit in the cache.



Supplied by CMU and also used in CS 33.

Quiz 1

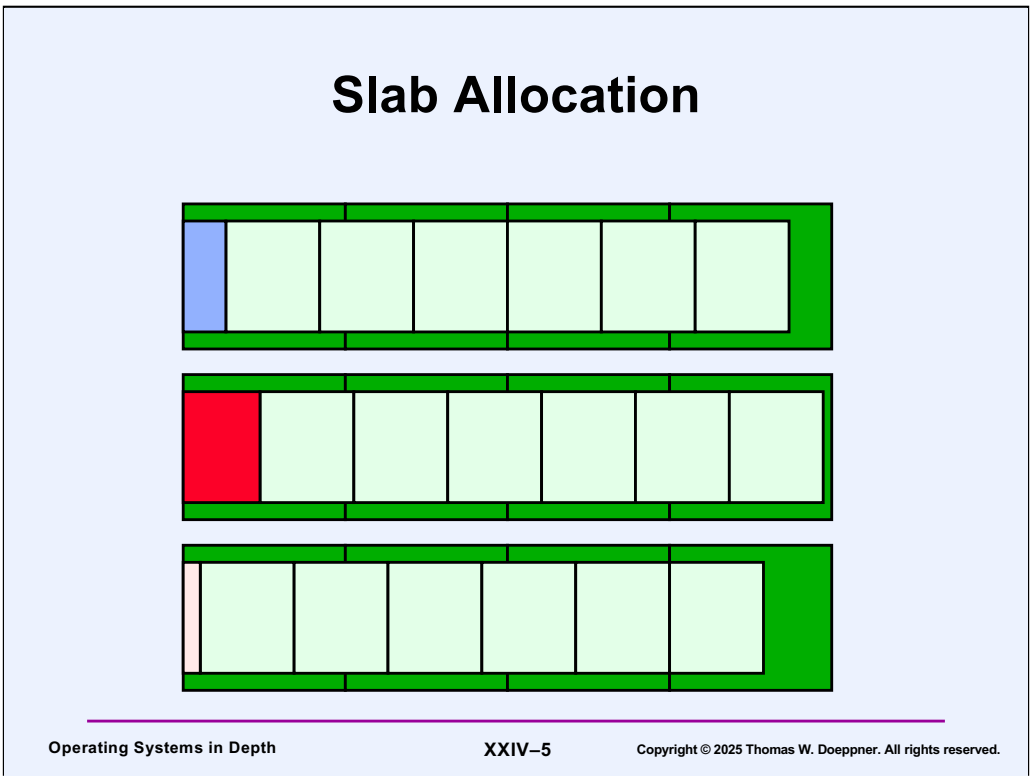
You're designing the algorithm for allocating an often-used and -allocated kernel data structure that fits within a cache line. We'd like to make sure that a number of these data structures can coexist in the hardware caches. Which one of the following would help make this happen (and is doable)?

a) Rounding the size of the data structure up to a power of 2

b) Making sure all reside in the same cache set

c) Making sure they are distributed across cache sets

d) Nothing would help



Many specific kinds of objects in the kernel, requiring less than a page of storage, are allocated and liberated frequently. Allocation involves finding an appropriate-sized area of storage, then initializing it, e.g. initializing various pointers as well as setting up synchronization data structures. Liberation involves tearing down the data structures and freeing the storage. If, for example the storage space is allocated strictly using the buddy system and the size of the objects are not a power of two, there's a certain amount of loss due to fragmentation. Further, there may be a fair amount of time overhead due to initialization and tearing down.

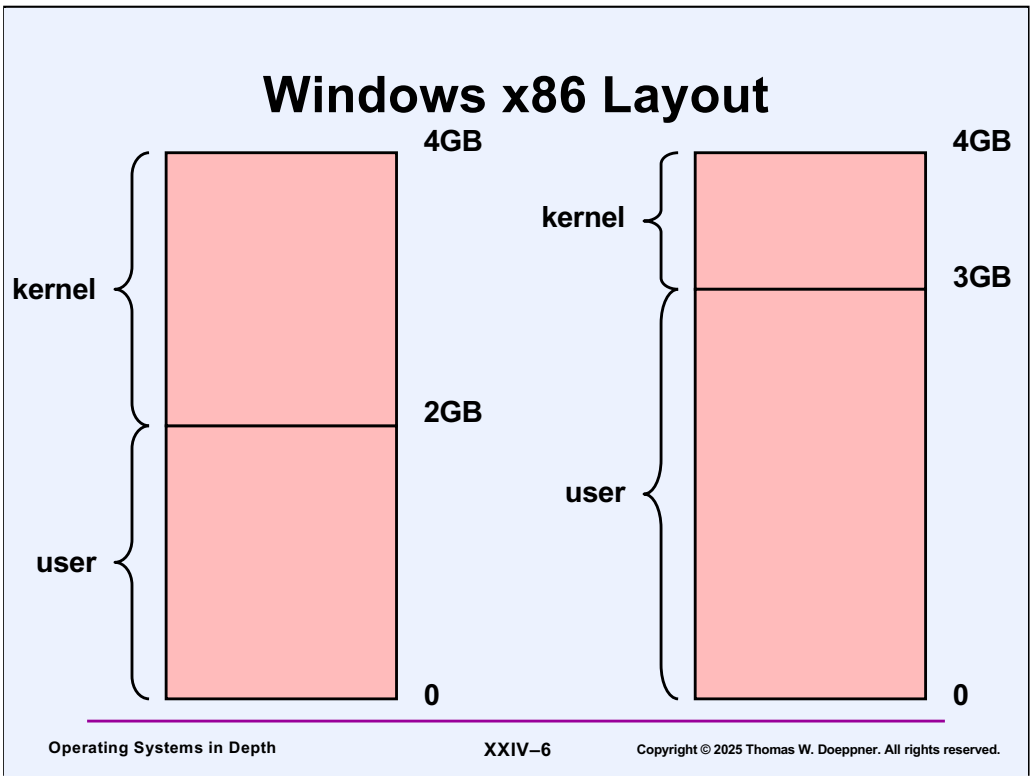
Slab allocation is a relatively new technique (developed at Sun: see "The Slab Allocator: An Object-Caching Kernel Memory Allocator," by Jeff Bonwick and appearing in the **Proceedings of the USENIX Summer 1994 Technical Conference**). For each type of object to be managed in such a fashion, a separate cache is set up. Contiguous sets of pages, called **slabs**, are allocated to hold objects. Whenever a slab is allocated, a constructor is called to initialize all the objects it holds. Then as objects are allocated, they are taken from the set of existing slabs in the cache. When objects are freed, they are simply marked as such, but made available for re-allocation without freeing their storage or tearing them down. Thus new objects can be allocated cheaply. When storage is given back to the kernel (either because there are too many free objects or the kernel requests storage due to a system-wide dearth of storage), entire slabs are returned, with each object in them appropriately torn down.

A further benefit of slab allocation is "cache coloring": if all instances of an object are aligned identically, then all will be occupying the same cache sets and thus only one (or a small number, depending on the cache) can be in the cache at once. However, by having successive slabs start the run of objects at different offsets from the beginning of

the slab, we arrange so that there are different “colors” of objects, each color with different alignment.

While it might be the case that each object is larger than a cache line, it's often true that the lower-addressed bytes are accessed more often that the other bytes. Thus the goal of cache coloring is to make sure that the addresses of these blocks cause there contents to be distributed across all cache sets, and thus their initial bytes will be distributed across all cache sets.

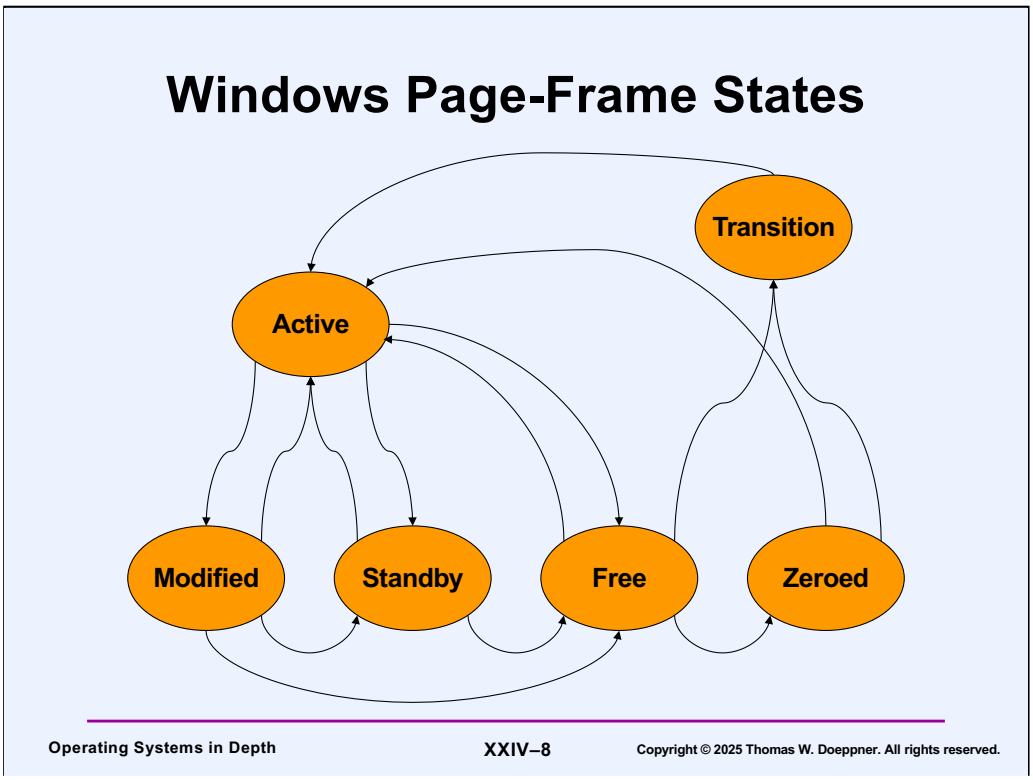
Slab allocation is used in Weenix.



Either arrangement may be configured.

Windows Paging Strategy

- All processes guaranteed a "working set"
  - lower bound on page frames
- Competition for additional page frames
- "Balance-set" manager thread maintains working sets
  - one-handed clock algorithm
- Swapper thread swaps out idle processes
  - first kernel stacks
  - then working set
- Some of kernel memory is paged
  - page faults are possible



See the text, Section 7.3.2.2 starting on page 308, for a detailed explanation.

Page frames in the **active** state hold pages that are in either a process's working set or the system's working set. The balance-set manager takes page frames out of this state and puts them into either the **standby** or **modified** states, corresponding to their being on the standby or modified lists. These lists contain threads that have not be used recently, but have not yet been made free (they are allowed to exist for a short additional period just in case there is a late reference to them). Two **page-writer threads** take page frames in the **modified** state, write them to secondary storage, and then put them in the **standby** state. The zero-page thread moves page frames from the **free** state to the **zeroed** state. When the contents

of a page frame are no longer useful the page frame is moved, by whatever thread discovers or causes the uselessness, to the **free** state. Pages are preferentially allocated from the free list. If that is empty, then they may be taken from the standby list.

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**

An efficient implementation of the **fork** system call was a challenge in the early implementations of Unix with virtual memory.

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* (or *exit*), the address space is handed back to the parent
- Advantages
  - very efficient
- Disadvantages
  - works only if child does an *exec* (or *exit*)
  - child shouldn't do anything to the address space

The first approach toward an efficient **fork** was the variant known as **vfork**, which first appeared in late 1979 as part of the third Berkeley Software Distribution (known as 3 BSD). It was remarkably efficient, but was not a complete solution (though good enough for most applications).

Quiz 2

Will the assertion evaluate to true?

```
volatile int A = 6;
…
if (vfork() == 0) {
  A = 7;
  exit(0);
}
sleep(1); // sleep for one second
assert(A == 7);
…
```

a) yes

b) no

Lazy Evaluation

- Always put things off as long as possible
- If you wait long enough, you might not have to do them

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
- Advantages
  - semantically equivalent to the original *fork*
  - usually faster than the original *fork*
- Disadvantages
  - slower than *vfork*

After the virtual-memory implementation matured, a better approach (in most respects) for a **fork** implementation became possible, using a technique known as **copy on write** (described in the next slides).

Quiz 3

How many pages of virtual memory must be copied from the parent to the child in the following code?

```
if (fork() == 0) {
  close(0);
  dup(open("input_file", O_RDONLY));
  execv("newprog", 0);
}
```

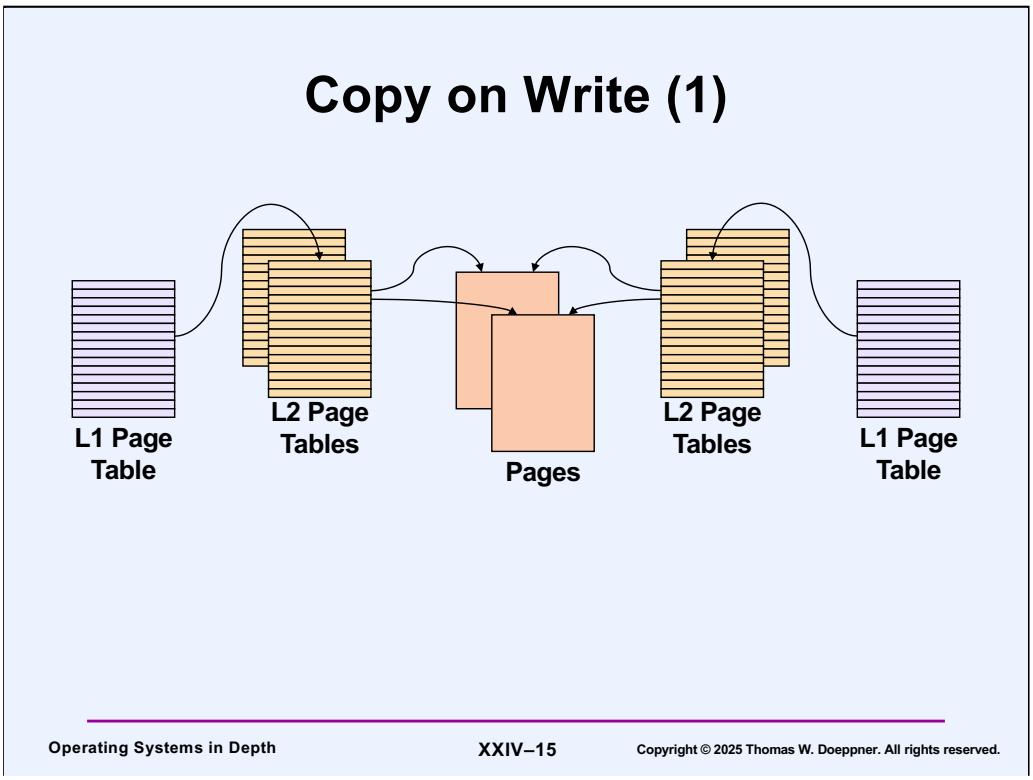
a) 0

b) 1-2

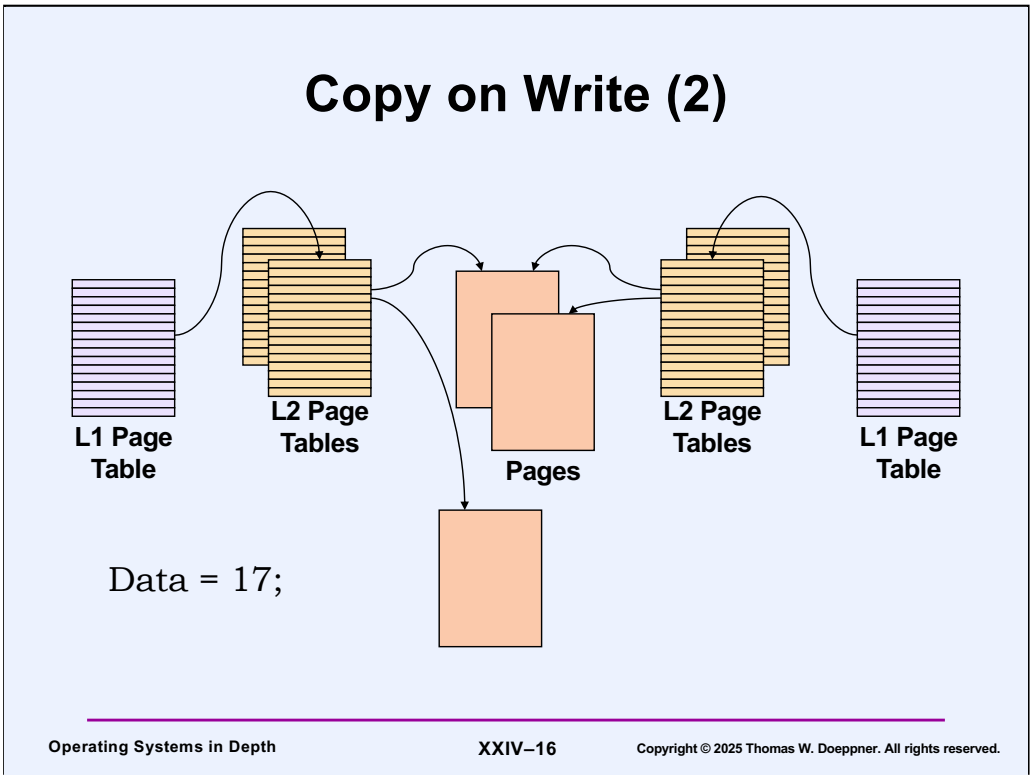
c) 4-8

d) lots

Assume a really good implementation of fork. Hint: keep in mind that we have to worry about text, data, and stack (among other possible regions).



In the copy-on-write approach, processes share their pages, which are marked read-only.



If a thread in either process attempts to modify a page, the hardware traps it, generating a protection fault. The operating system realizes that an attempt is being made to modify a page marked copy on write, so it makes a copy of that page for the faulting process. The appropriate page table is modified to point to the new page. Note that we could apply copy on write to the L2 page tables as well, so as to avoid having to copy them.

Thus we are postponing copying the address space, in hopes that we might not have to do it.

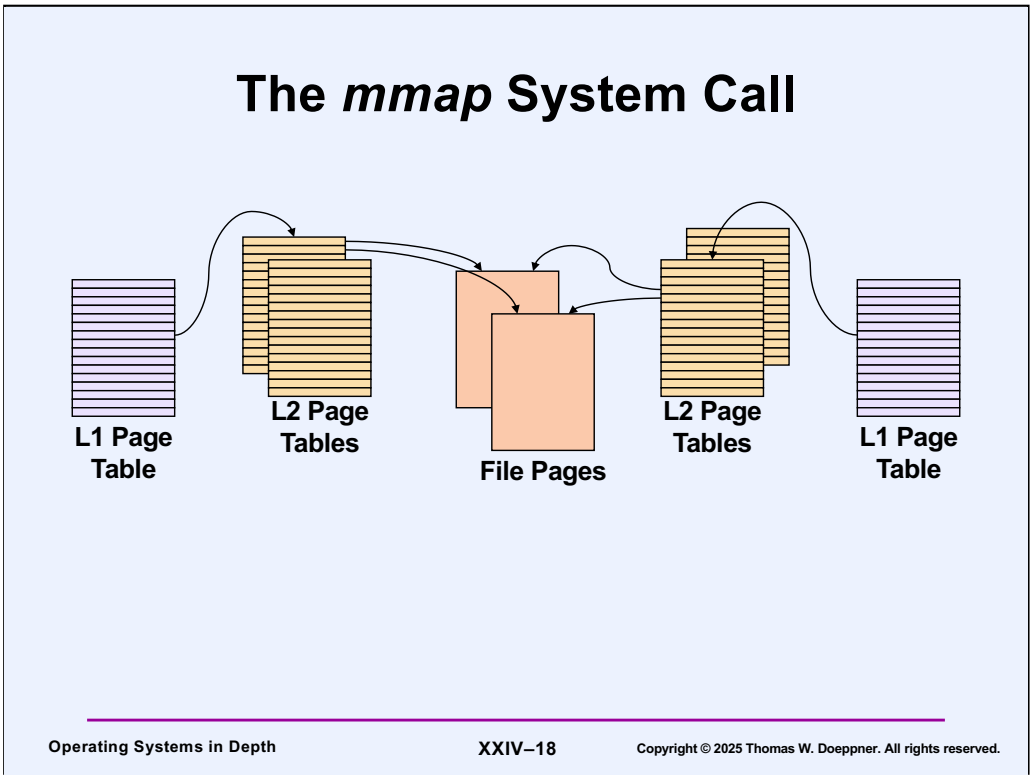
Quiz 4

We have a file that contains one billion 64-bit integers. We are writing a program to read in the file and add up all the integers. Which approach will be fastest:

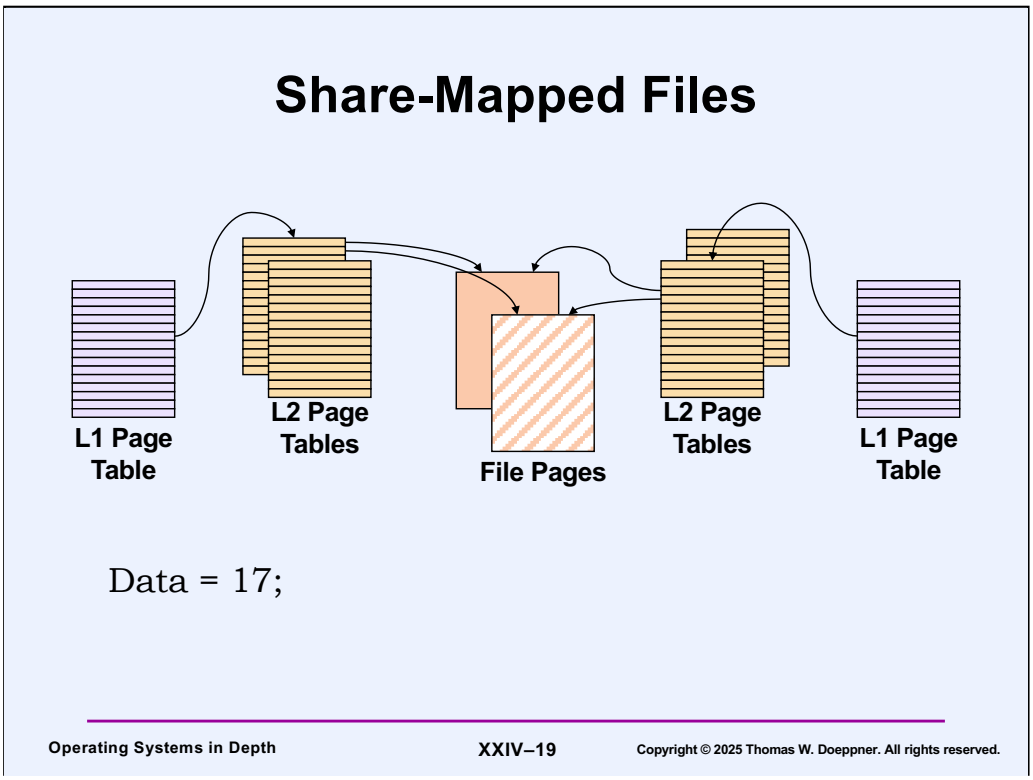
a) read the file 8 bytes at a time, adding to a running total what is read in

b) read the file 8k bytes at a time, then add each of the integers contained in that block to the running total

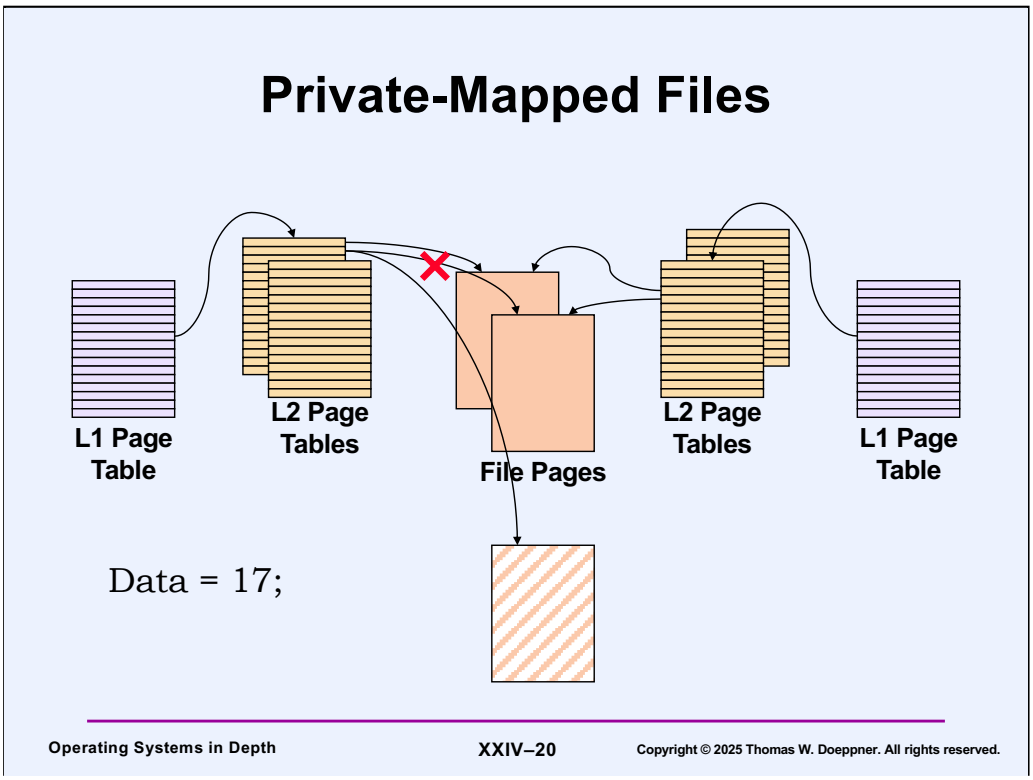
c) ***mmap*** the file into the process's address space, then sum up all the integers in this mapped region of memory



The **mmap** system call maps a file into a process's address space. All processes mapping the same file can share the pages of the file.

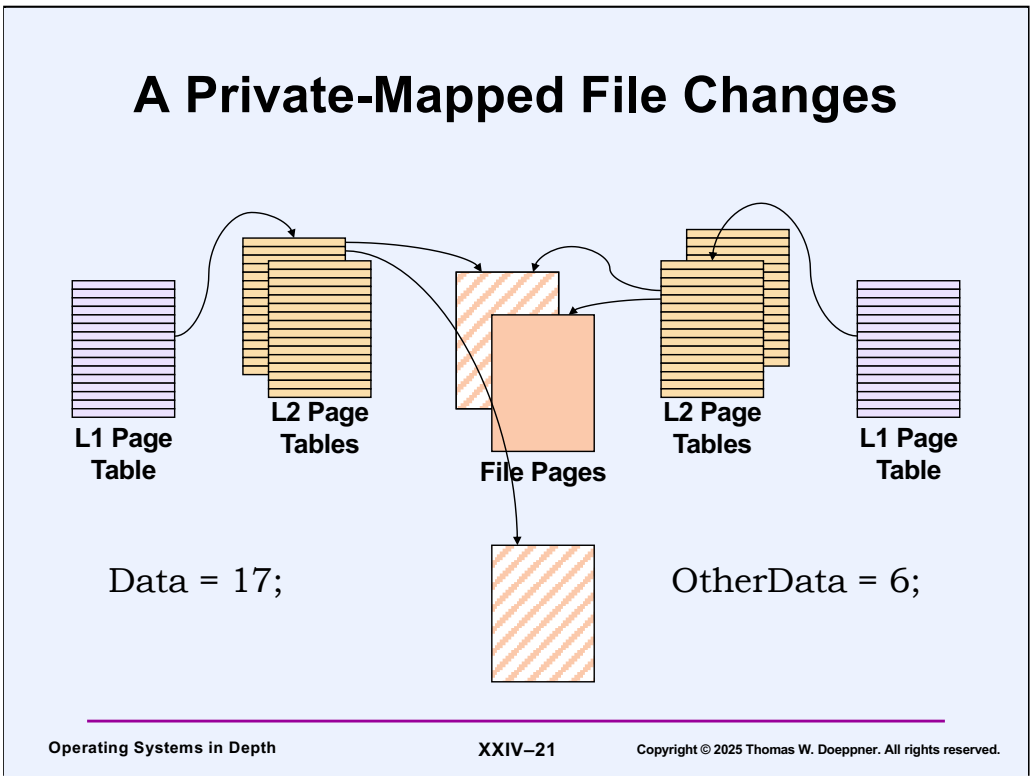


There are a couple options for how modifications to mmapped files are dealt with. The most straightforward is the **share** option in which changes to mmapped file pages modify the file and hence the changes are seen by the other processes who have share-mapped the file.



The other option is to **private-map** the file: changes made to mmapped file pages do not modify the file. Instead, when a page of a file is first modified via a private mapping, a copy of just that page is made for the modifying process, but this copy is not seen by other processes, nor does it appear in the file.

In the slide, the process on the left has private-mapped the file. Thus its changes to the mapped portion of the address space are made to a copy of the page being modified.



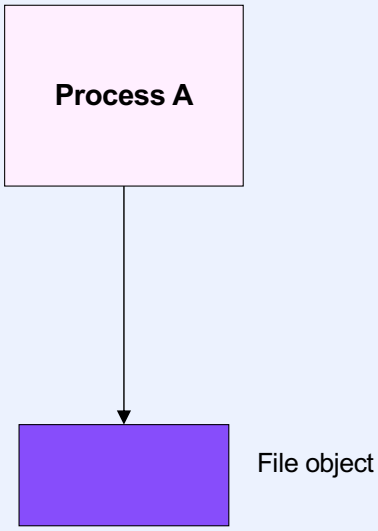
What if a private-mapped file is changed by a process that has share-mapped it? Somewhat surprisingly, the change is seen by the process that has private-mapped the file, as long as the change is not to a page that has been modified in the private-mapped view.

In the slide, while the process on the left has private-mapped the file, the one on the right has share-mapped it. Thus changes by the process on the right to pages that have not been modified by the process on the left are seen by both processes.

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

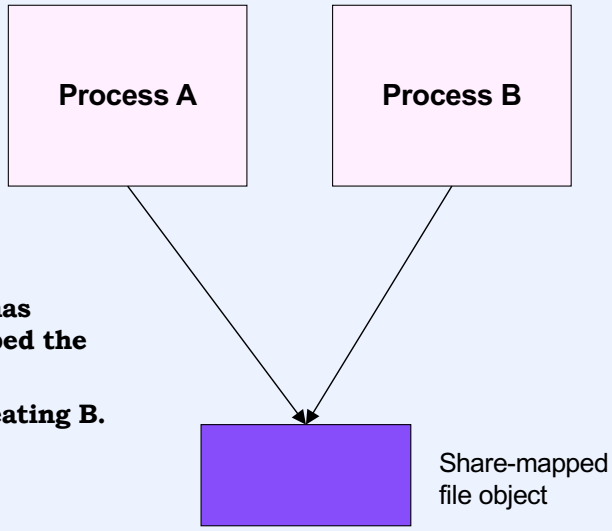
Share Mapping (1)



Process A has share mapped the file object.

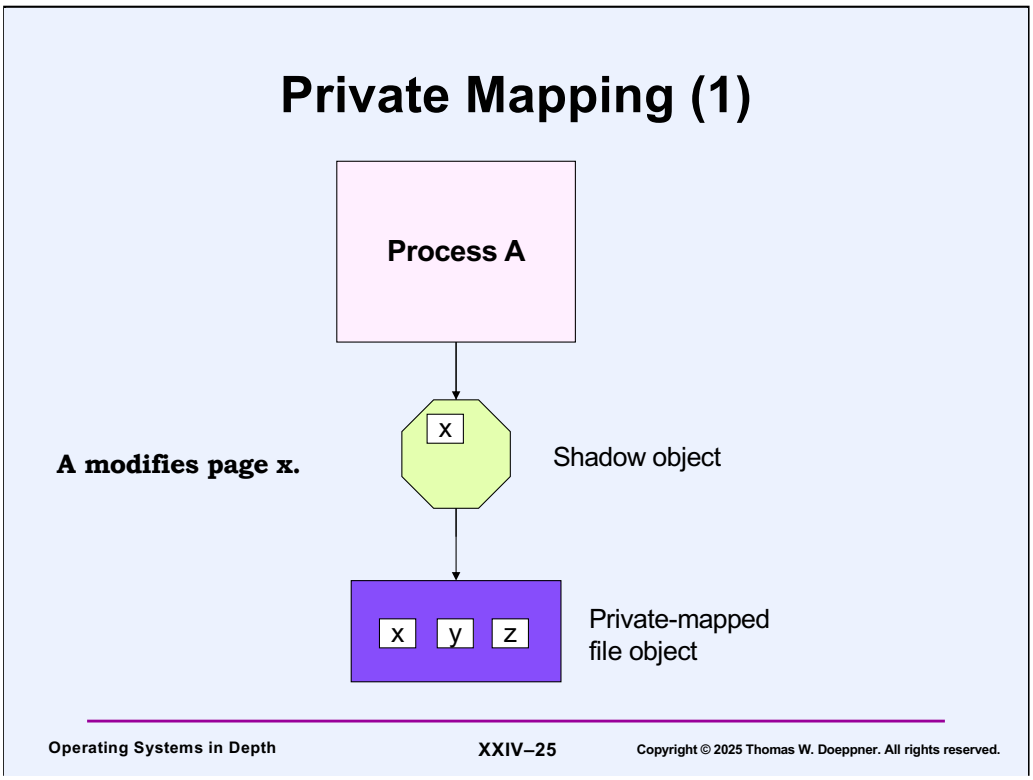
File object

Share Mapping (2)

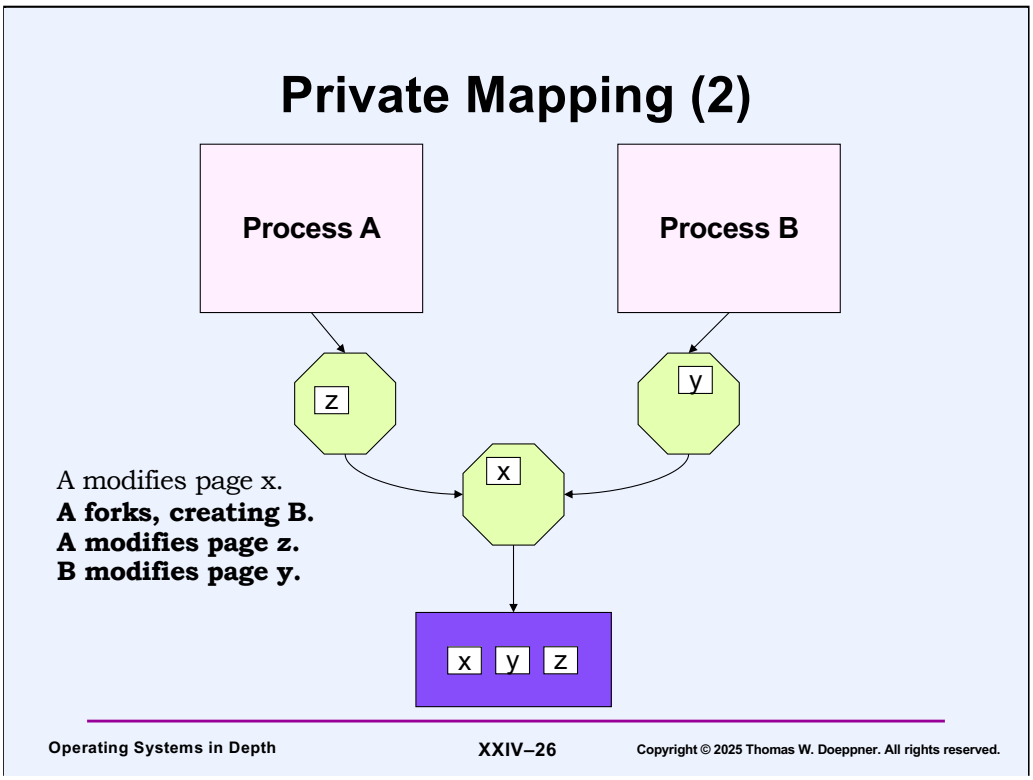


Process A has share-mapped the file object.

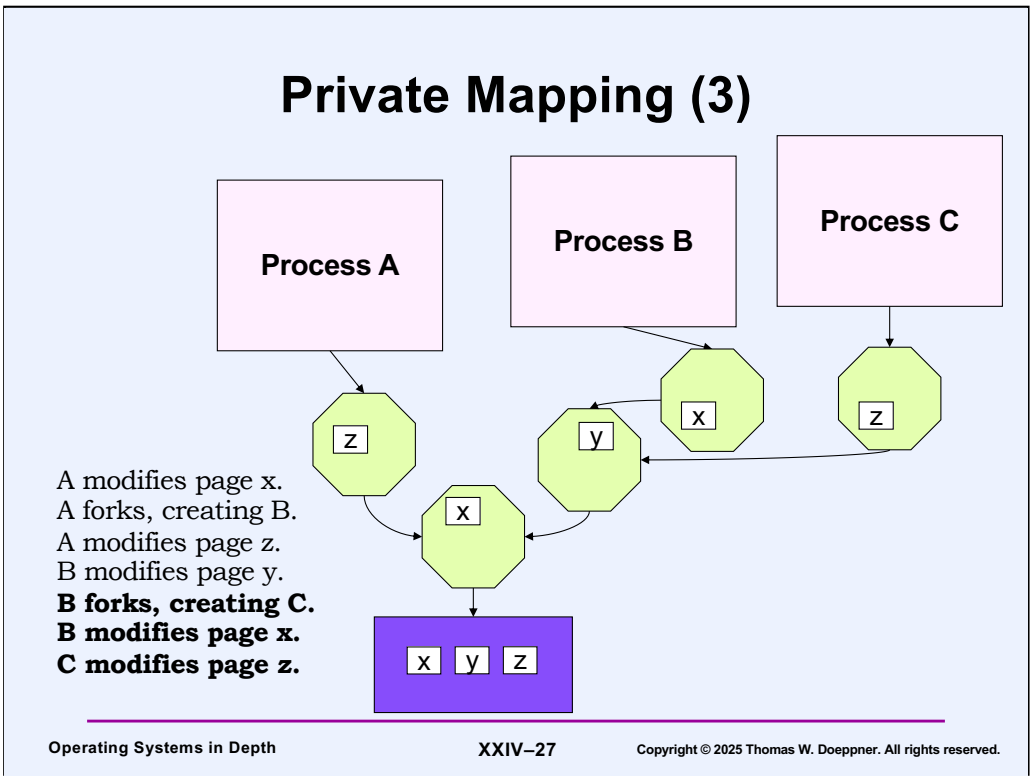
A forks, creating B.



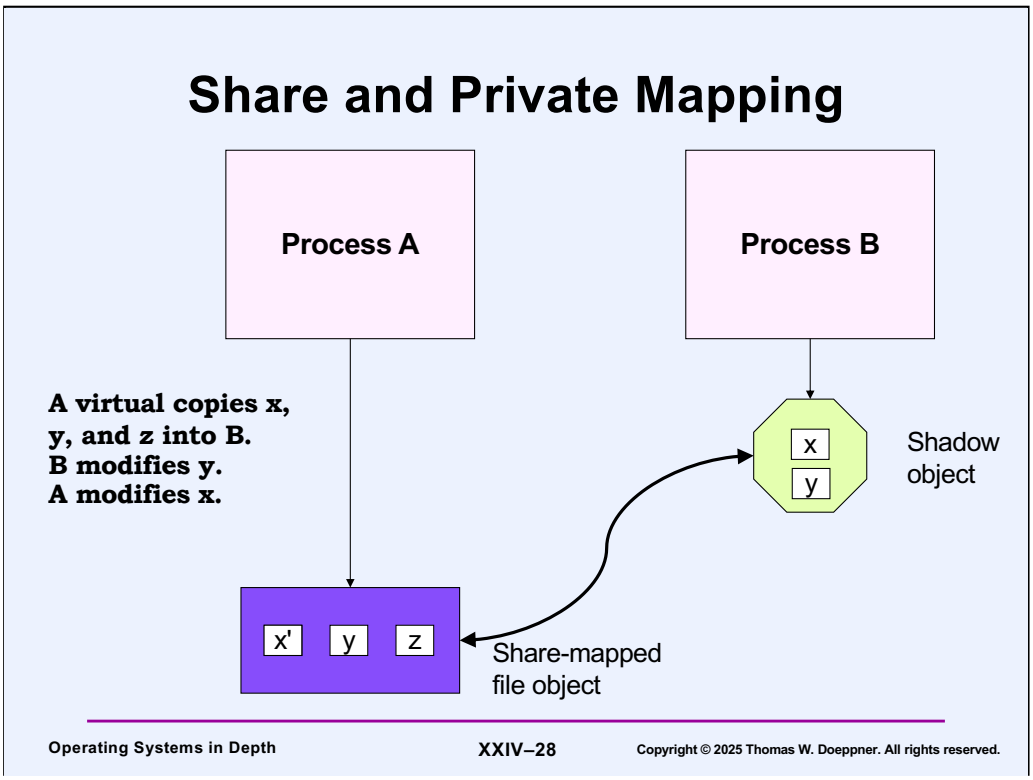
As an example of how to implement virtual copy and private-mapping operations efficiently, we look at the approach used by the Mach operating system and adopted by Apple in Mac OS X (and adopted by us for Weenix). Suppose process A has a private mapping of the file and has modified page x but neither page y nor page z. Since the file is private-mapped, a copy of x is created in what's known as a **shadow object**. If it is necessary to page out x, it will be paged out to storage set up for this shadow object.



Process A forks, creating child process B. Shadow objects are created to contain the further modified pages of A as well as of B. B modifies page y and A modifies page z. If A accesses page y, it will obtain the copy in the original file, not the copy in the shadow object created for B. Similarly, if B accesses z, it will obtain the copy in the original file. If either process accesses x, they will both obtain the copy in the first shadow object.



Process B forks, producing child C. B modifies page x; C modifies page z.



Process A has share-mapped a file, then performs a virtual copy of the mapped portion of its address space into Process B. Thus B now has a private mapping of the file object. If B modifies page y, then, as before, it first makes a copy of y in its shadow object. If A modifies page x, the page is first copied to B's shadow object and then is modified in the file. (Note: since neither Linux nor Weenix supports the virtual-copy operation, this scenario does not occur in them. However, it does occur in systems that support virtual-copy operations efficiently)

Quiz 5

Unix process X has private-mapped a file into its address space. Our system has one-byte pages and the file consists of four pages. The pages are mapped into locations 100 through 103. The initial values of these pages are all zeroes.

1) X stores a 1 into location 100
2) X forks, creating process Y
3) X stores a 1 into location 101
4) Y stores a 2 into location 102
5) Y forks, creating process Z
6) X stores 111 into location 100
7) Y stores 222 into location 103
8) Z sums the contents of locations 100, 101, and 102, and stores them into location 103

What value did Z store into 103?

Answer:

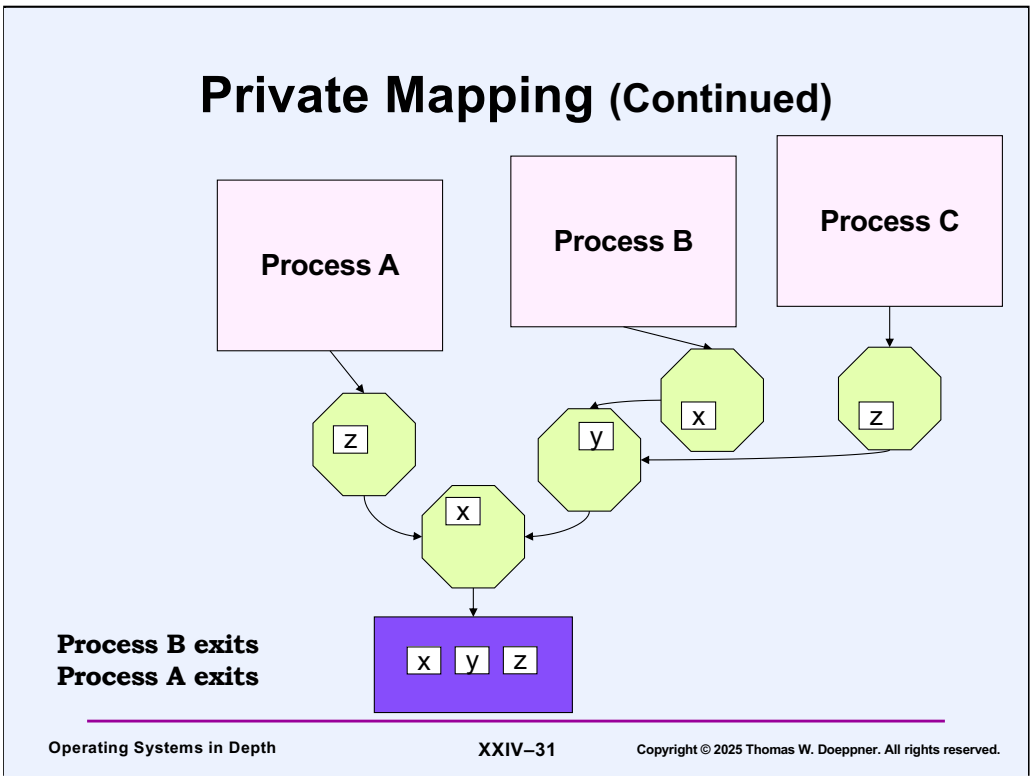
a) 0
b) 3
c) 4
d) 113

Fork Bomb!

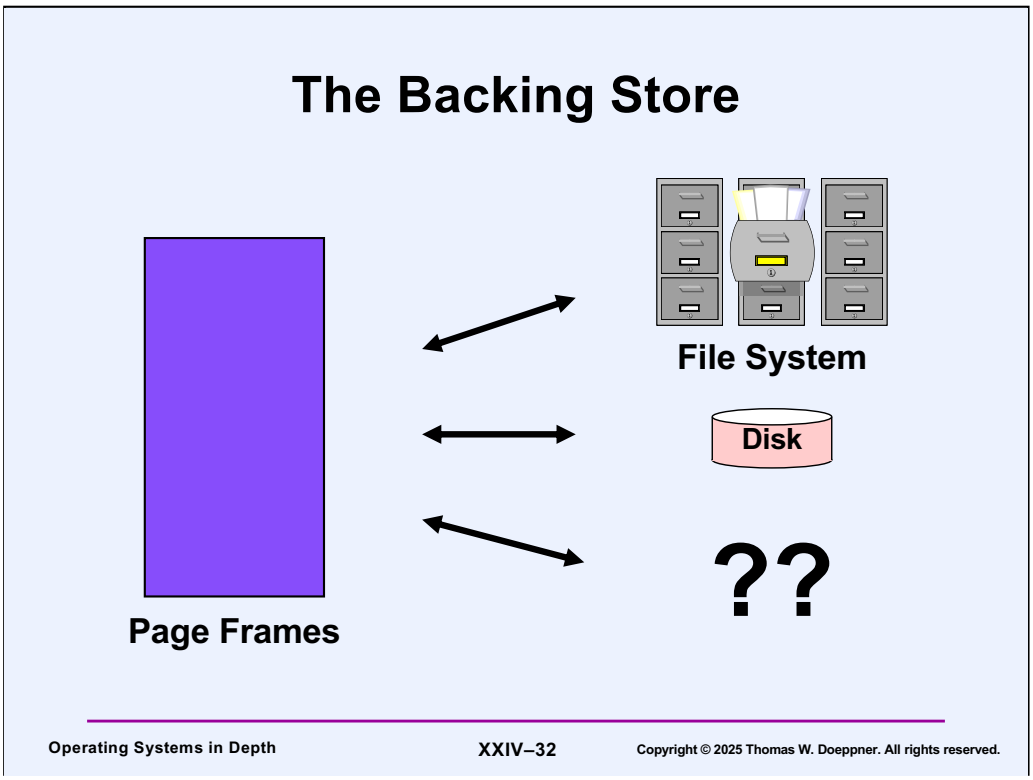
```
int main() {
  while (1) {
    if (fork() <= 0)
      exit(0);
  }
  return 0;
}
```

```
int main() {
  while (1) {
    if (fork() > 0)
      exit(0);
  }
  return 0;
}
```


These programs should run forever. For them to do so, some trimming must be done of the list of shadow objects.



After processes B and A exit, which shadow objects can be eliminated?



Our next topic is the backing store, i.e., the storage where pages are kept when not in primary memory. As shown in the slide, such storage might be managed by the file system, it might be some otherwise unstructured portion of a disk, or perhaps something else.

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

This slide and the next list the various possibilities of the backing-store location for Unix.

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)
  - pages are created as *zero fill on demand*; they must be backed up in swap space

Swap Space

- Space management possibilities
  - radical-conservative approach: pre-allocation
    - backing-store space is allocated when virtual memory is allocated
    - page outs always succeed
    - might need to have much more backing store than needed
  - radical-liberal approach: lazy evaluation
    - backing-store space is allocated only when needed
    - page outs could fail because of no space
    - can get by with minimal backing-store space

Equally important is when and how backing-store space is allocated.

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

With the "maybe" approach, various heuristics are used to come up with a good guess as to a reasonable level of over-commitment, much like how airlines over-commit seats on planes. And as is the case with airlines, things don't always work out. So, it's possible that a process may have to be killed if it cannot continue without additional memory (either primary storage (RAM) or swap space). Note that such a process isn't necessarily the "culprit". A large process may have used up most of the available resources. The kernel is trying to write out a page belonging to a smaller process. If there's no room for it in swap space (on disk), then it's the smaller process that's terminated.

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