File Systems Part 7

Scenarios

- Power failure at inopportune moment
  - "live data" is not modified
  - single lost write can be recovered
- Obscure bug in controller firmware or OS
  - detected by checksum in pointer
- Sysadmin accidentally scribbled on one drive
  - detected and repaired
- Out of disk space
  - add to the pool; SPA will cope
- Out of address space
  - 2^{128} is big
    - 1 address per cubic yard of a sphere bounded by the orbit of Neptune

More from ZFS

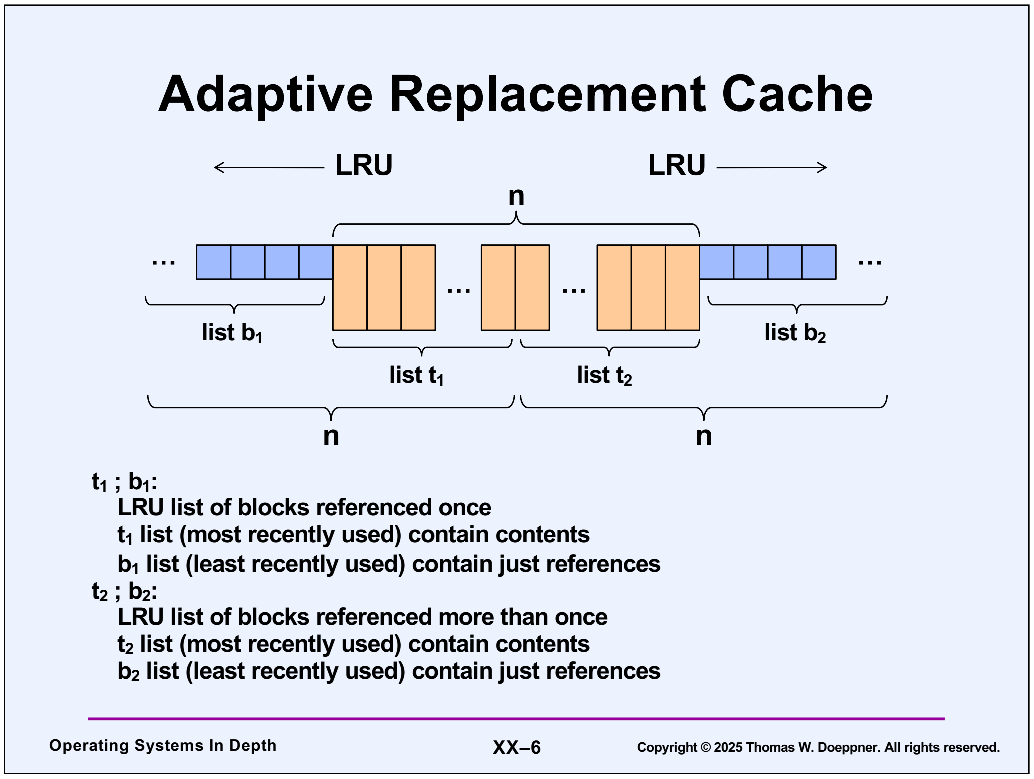
- Adaptive replacement cache
- Advanced prefetching

LRU Caching

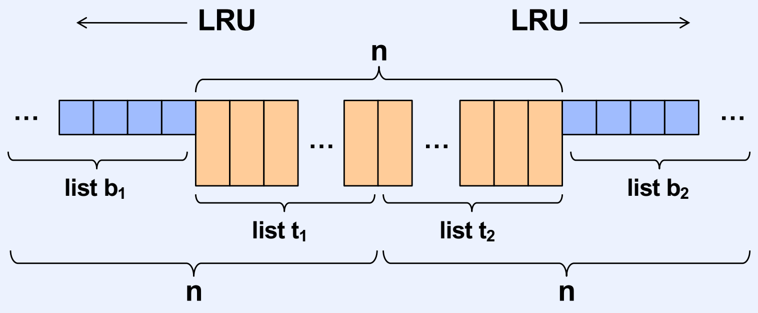
- LRU cache holds n least-recently-used disk blocks
  - working sets of current processes
- New process reads n-block file sequentially
  - cache fills with this file's blocks
  - old contents flushed
  - new cache contents never accessed again

(Non-Adaptive) Solution

- Split cache in two
  - half of it is for blocks that have been referenced exactly once
  - half of it is for blocks that have been referenced more than once
- Is 50/50 split the right thing to do?



Adaptive Replacement Cache

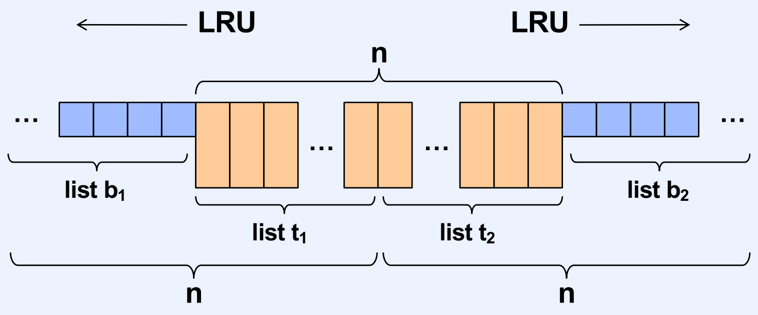


t_1 ; b_1:
- LRU list of blocks referenced once
- t_1 list (most recently used) contain contents
- b_1 list (least recently used) contain just references

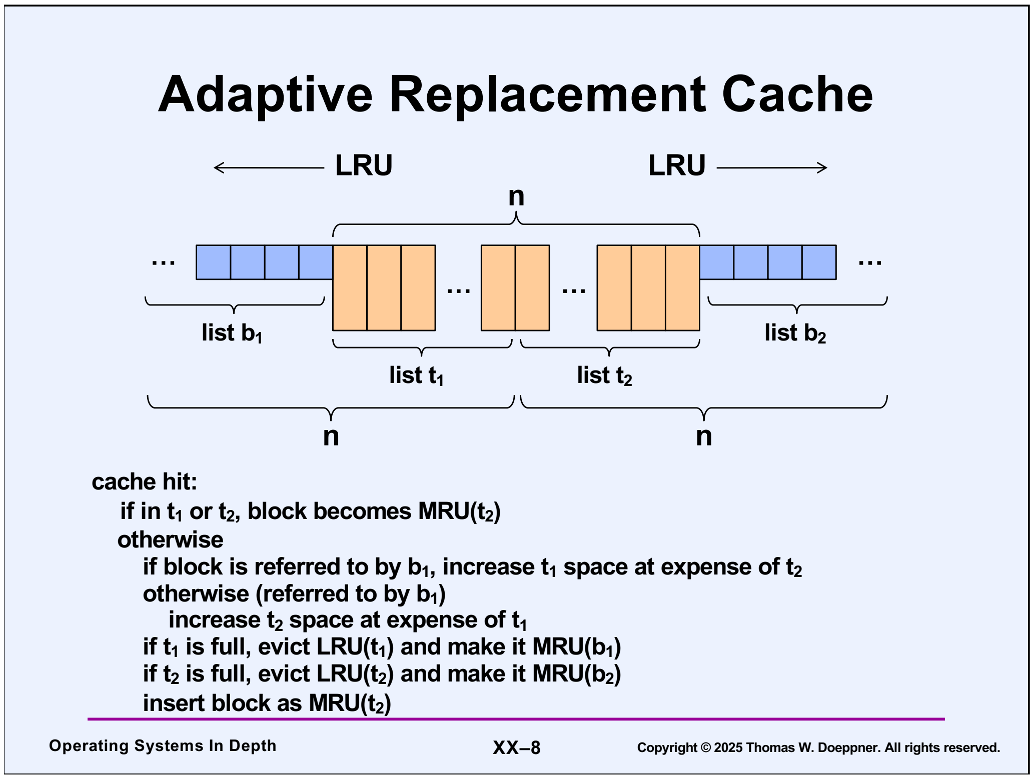
t_2 ; b_2:
- LRU list of blocks referenced more than once
- t_2 list (most recently used) contain contents
- b_2 list (least recently used) contain just references

A description of the algorithm can be found at http://www.usenix.org/publications/login/2003-08/pdfs/Megiddo.pdf. A more detailed paper that includes an analysis can be found at http://www.usenix.org/publications/library/proceedings/fast03/tech/full_papers/megiddo/megiddo.pdf.

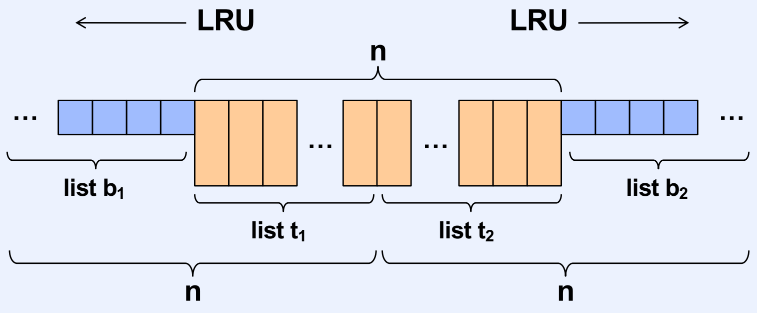
Adaptive Replacement Cache



```
cache miss:
  if $t_1$ is full
        evict LRU($t_1$) and make it MRU($b_1$)
  referenced block becomes MRU($t_1$)
```



Adaptive Replacement Cache



```
cache hit:
  if in $t_1$ or $t_2$, block becomes MRU($t_2$)
  otherwise
    if block is referred to by $b_1$, increase $t_1$ space at expense of $t_2$
    otherwise (referred to by $b_1$)
      increase $t_2$ space at expense of $t_1$
    if $t_1$ is full, evict LRU($t_1$) and make it MRU($b_1$)
    if $t_2$ is full, evict LRU($t_2$) and make it MRU($b_2$)
    insert block as MRU($t_2$)
```

Note that the size of the combined t_1 and t_2 lists doesn't change – it's n. Thus if there was a cache hit and the block was in t_1, it moves to t_2, increasing the size of t_2 and decreasing the size of t_1.

Quiz 1

Lists b_1 and b_2 do not contain cached blocks, but just their addresses. Why are they needed?

a) So that one can determine how much better things would be if the cache were twice as large

b) As placeholders so that when these blocks are read in, it's known where in the cache they would go

c) So that we would know, if the addressed block is referenced, whether it would have been in the cache if the corresponding t list were larger

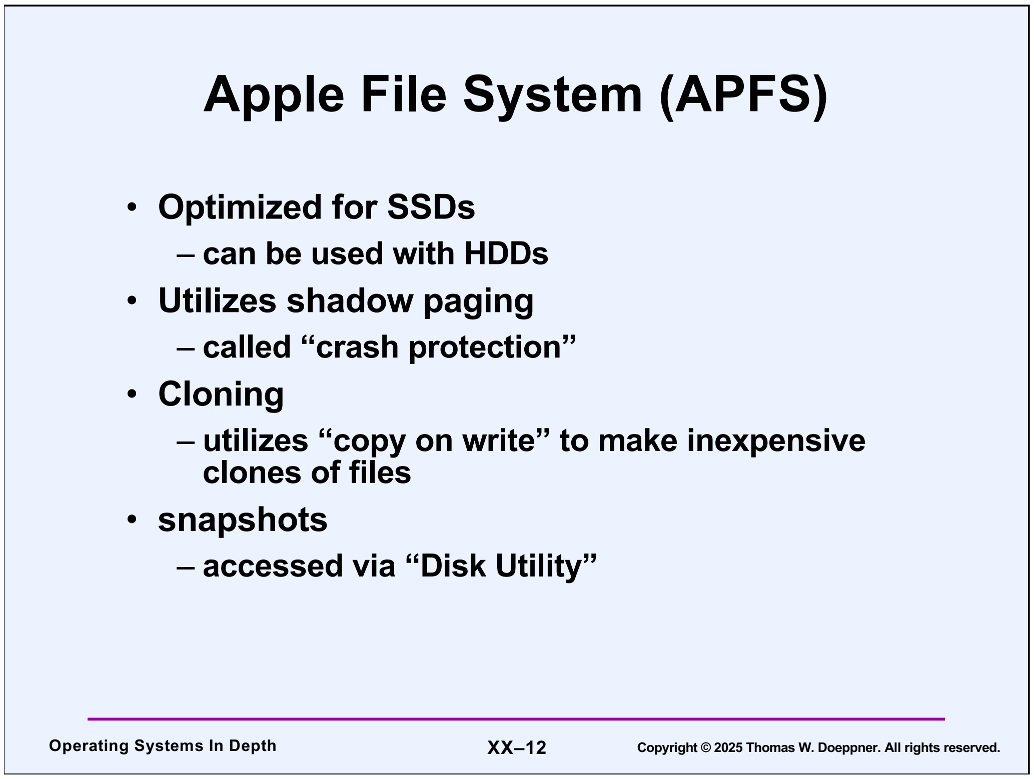
d) They are used by the file system to help determine block reference patterns

Prefetch

- FFS prefetch
  - keeps track of last block read by each process
  - fetches block i+1 if current block is i and previous was i-1
  - chokes on
    - diff file1 file2

zfetch

- Tracks multiple prefetch streams
- Handles four patterns
  - forward sequential access
  - backward sequential access
  - forward strided access
    - iterating across columns of matrix stored by columns
  - backward strided access


Apple File System (APFS)

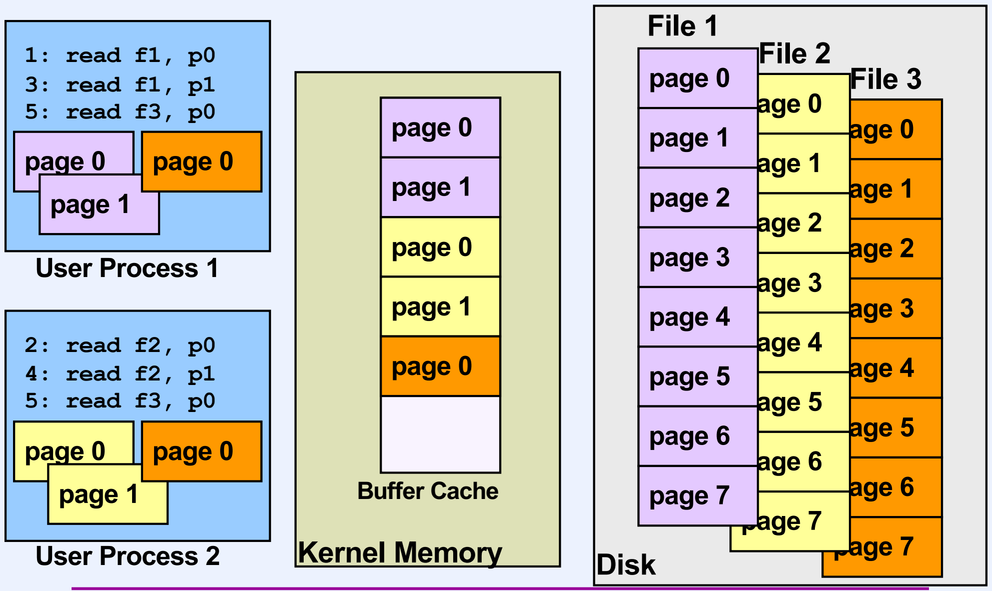
- Optimized for SSDs
  - can be used with HDDs
- Utilizes shadow paging
  - called “crash protection”
- Cloning
  - utilizes “copy on write” to make inexpensive clones of files
- snapshots
  - accessed via “Disk Utility”

APFS was released in March 2017.

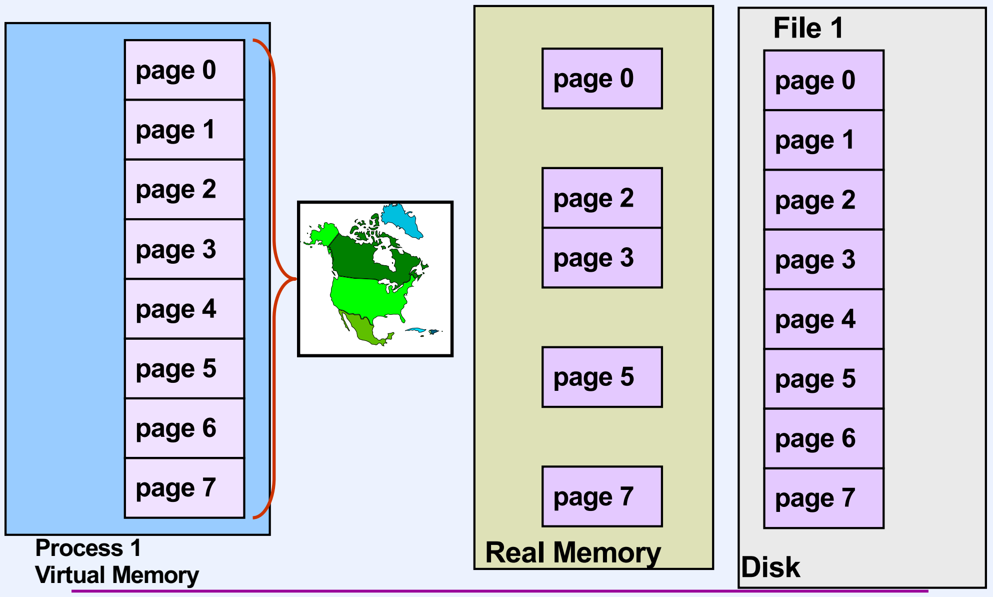
File System Implementation Concerns

- System-call access to files vs. mmap access
  - VFS integration with virtual memory
- File-based block indexing vs. file-system-based block indexing
- File-system block size vs. page size
  - conveniently identical on Weenix (4096 bytes)

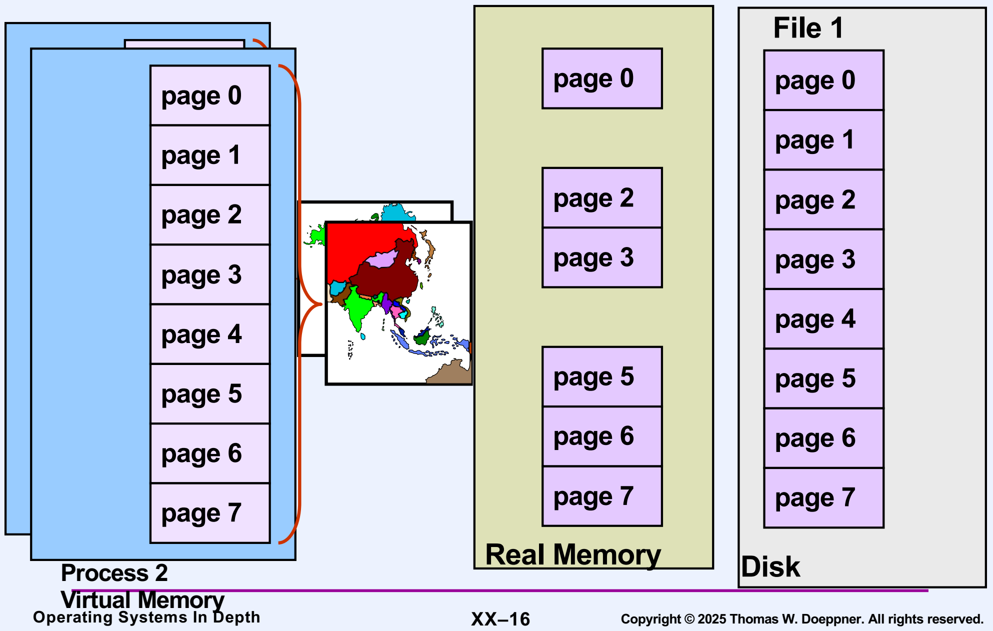
Traditional I/O

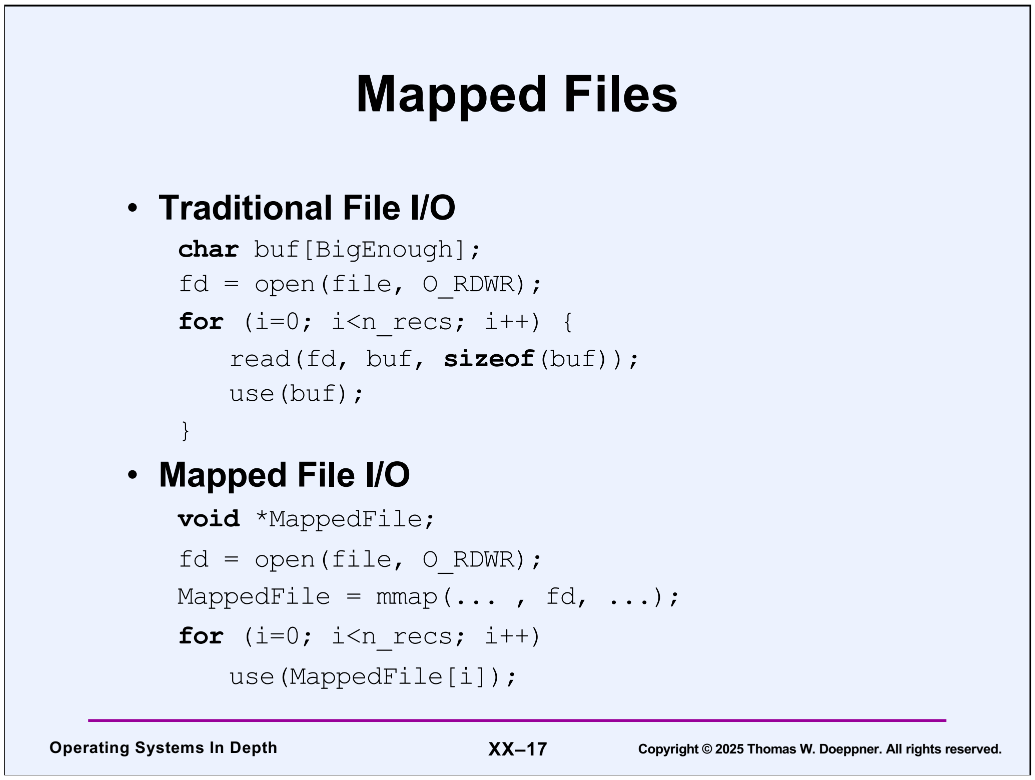


Mapped File I/O



Multi-Process Mapped File I/O





Mapped Files

- **Traditional File I/O**

```
char buf[BigEnough];
fd = open(file, O_RDWR);
for (i=0; i<n_recs; i++) {
    read(fd, buf, sizeof(buf));
    use(buf);
}
```

- **Mapped File I/O**

```
void *MappedFile;
fd = open(file, O_RDWR);
MappedFile = mmap(... , fd, ...);
for (i=0; i<n_recs; i++)
    use(MappedFile[i]);
```

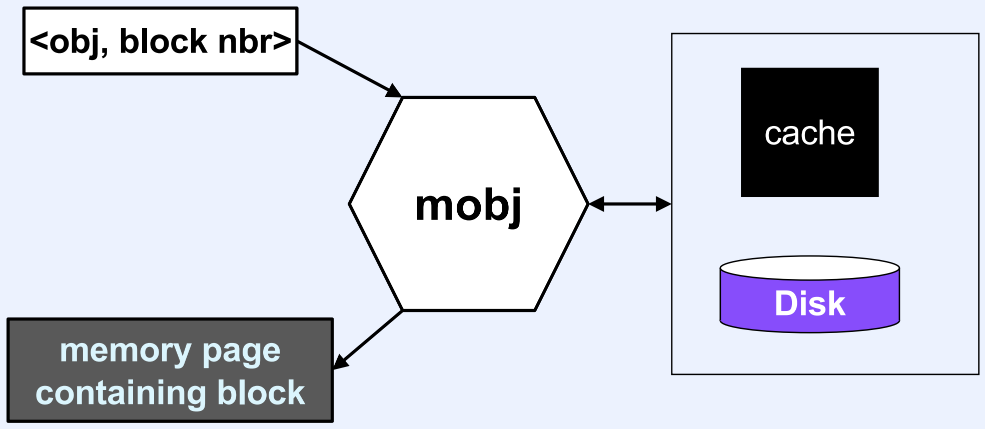
Traditional I/O involves explicit calls to read and write, which in turn means that data is accessed via a buffer; in fact, two buffers are usually employed: data is transferred between a user buffer and a kernel buffer, and between the kernel buffer and the I/O device.

An alternative approach is to **map** a file into a process's address space: the file provides the data for a portion of the address space and the kernel's virtual-memory system is responsible for the I/O. A major benefit of this approach is that data is transferred directly from the device to where the user needs it; there is no need for an extra system buffer.

Consistency

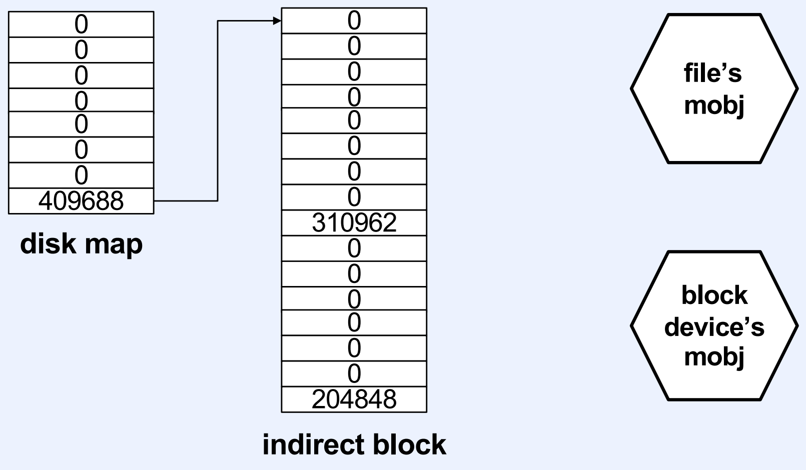
```
typedef struct
    {int flags; char morestuff[OSIZE];} object_t;
object_t object, *mregion;
int fd;
int buf;
fd = open("file", O_RDWR);
mregion = (object_t *)mmap(0, sizeof(object),
    PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
buf = 6;
write(fd, &buf, sizeof(buf));
if (mregion->flags != 6)
  fprintf(stderr, "something is wrong!\n");
```

Memory Objects



A memory object (mobj) is an abstraction implemented in the OS kernel that, given a block number within some sort of object (say a file or a disk), finds the block if it exists (in either the cache or the on-disk file system) and returns a pointer to a copy of it in kernel memory. How it works is, of course, very dependent on the virtual-memory subsystem and the type of file system.

Memory Objects and Files



Here we have the disk map of an S5FS file, which is contained in its inode. In the Weenix version of S5FS there are no doubly or triply indirect blocks, just a single indirect block, which is shown in the slide (at address 409688). No data has been written to the direct portion of the file, but data in two blocks have been written in the indirect portion of the file, in blocks whose addresses are 310962 and 204848. Thus all the other blocks of the file are treated as if they contain zeroes, even though they are not (currently) stored on disk.

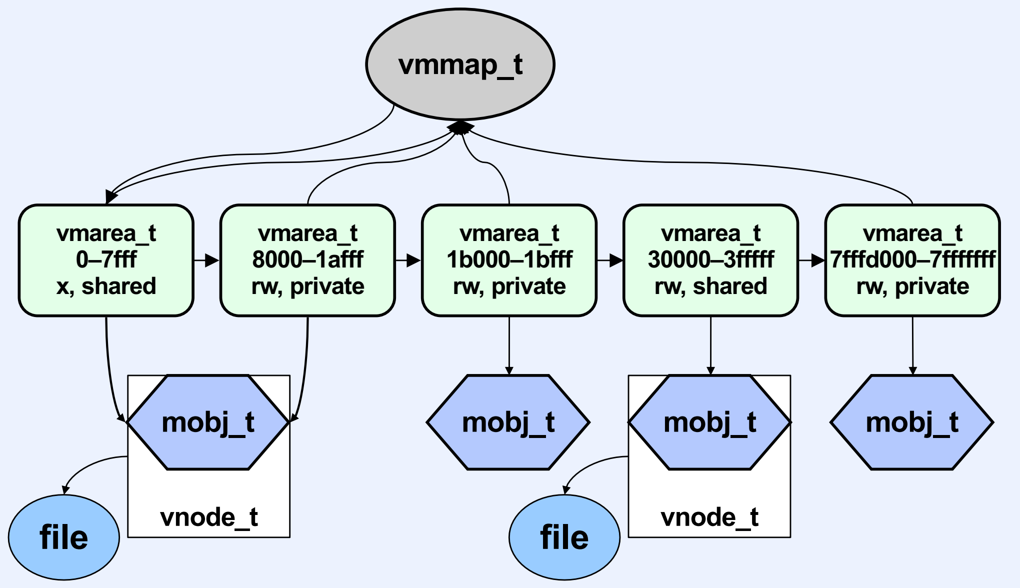
The file system resides on disk, which is represented in Weenix as a **block device** (it's accessed via blocks).

The file’s mobj is used to find data blocks of the file. Thus, given a data-block number, it will be used to find the block if it exists (in either the cache or the on-disk file system) and return a pointer to a copy of it in kernel memory.

But for indirect blocks and free blocks, which aren’t in the data space of a file, the block device’s mobj is used to find them. In these cases, the block numbers are block numbers within the block device, not within a file.

So as to keep track of the which blocks are cached in kernel memory, each mobj has a list of page frames (physical memory) holding cached blocks for its associated object (whether file or block device).

Weenix Address Space Rep



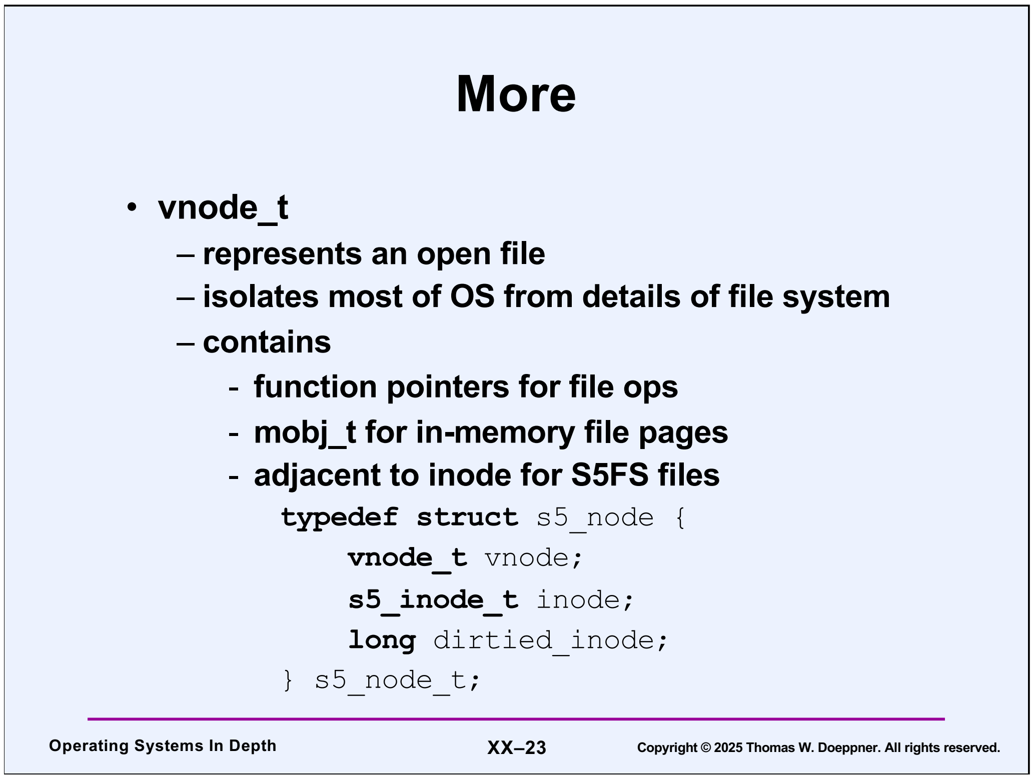
Here we again see the Weenix address space representation for a process, this time with some detail about the objects that are mapped into it. The two leftmost vmarea structs represent text and initialized data. They have the same file mapped into them, just different portions of the file for each. The file is represented by a vnode, which contains an mobj (of type mobj_t).

The middle vmarea struct represents uninitialized data (BSS) and the dynamic area. Though it's initialized with zeroes, it's not associated with any file. It's represented by an mobj structure that's marked as being **anonymous** – meaning there is no name for it in the file system, and thus it has no inode. This is also true for the rightmost vmarea struct, which is for the stack.

The remaining vmarea struct represents a file that has been mapped into the address space and is shared. Representing it is both an mobj and a vnode.

Some Data Structures

- pframe_t
  - represents a page frame
    - points to actual frame
    - refers to frame in lists
- mobj_t
  - refers to list of in-memory pages (page frames) of an object such as a file
  - page frames represented by pframe_t's
- vmarea_t
  - represents a region within an address space
  - into which an object is mapped
    - represented by an mobj_t



More

- **vnode_t**
  - represents an open file
  - isolates most of OS from details of file system
  - contains
    - function pointers for file ops
    - mobj_t for in-memory file pages
    - adjacent to inode for S5FS files

```
typedef struct s5_node {
    vnode_t vnode;
    s5_inode_t inode;
    long dirtied_inode;
} s5_node_t;
```

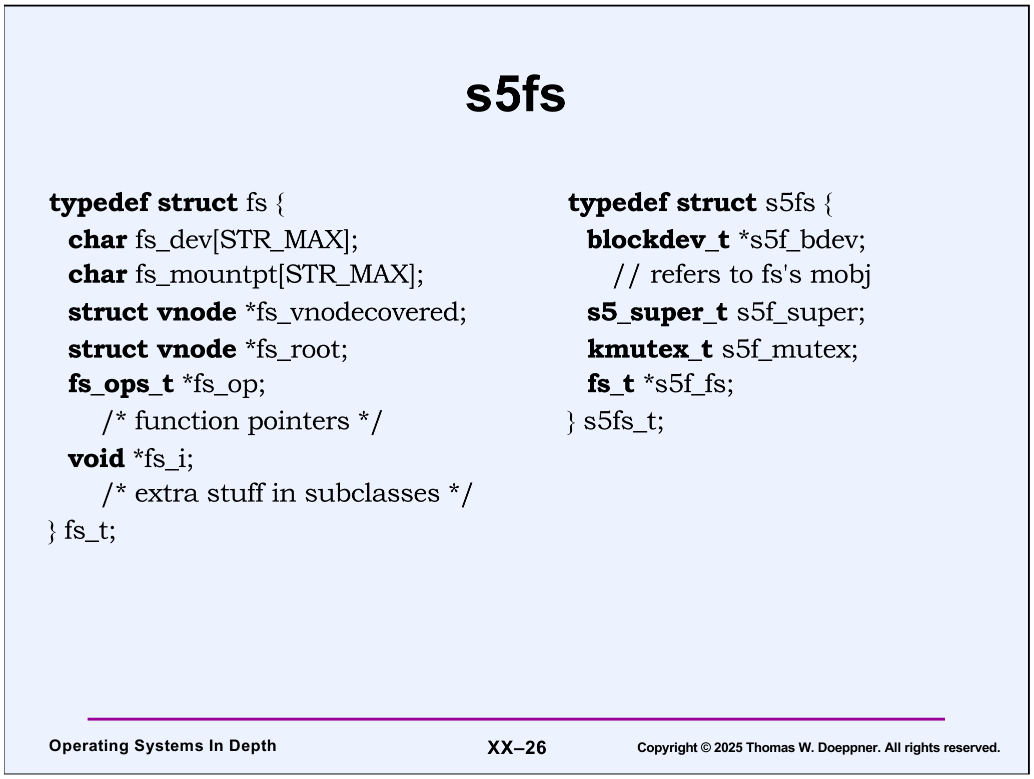
An s5_node_t object contains both the vnode and a copy of the inode (from disk). It also contains a flag (represented as a long) that indicates whether the inode has been modified (made "dirty"). If so, it must eventually be written to disk.

vnode

```
typedef struct vnode {
   unsigned short vn_refcount;
   struct fs *vn_vfsmounted;
   struct fs *vn_vfs;
   unsigned long vn_vno;
   int vn_mode;
   int vn_len;
   link_list_t vn_link;
   kmutex_t vn_mutex;
   struct vnode_ops *vn_op;
      /* function pointers */
   mobj_t mobj;
   void *vn_i;
      /* extra stuff in subclasses */
} vnode_t;
```

Caching

- A file’s list of cached pages is in its mobj_t
- System-call file access
  - get to file’s mobj_t via vnode_t, and then to block device’s mobj_t
- Mmap file access
  - get to mobj_t via vmarea_t



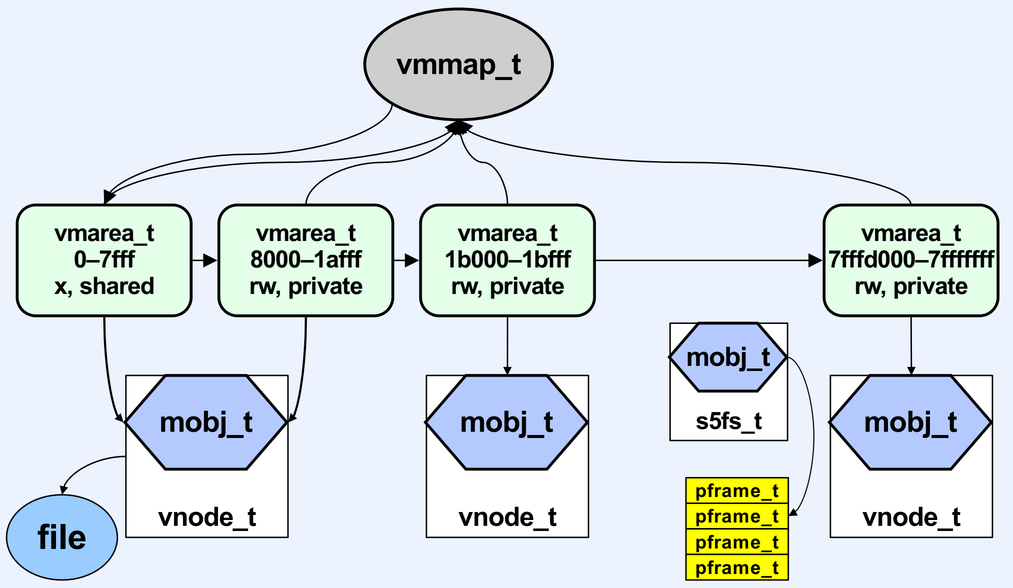
s5fs

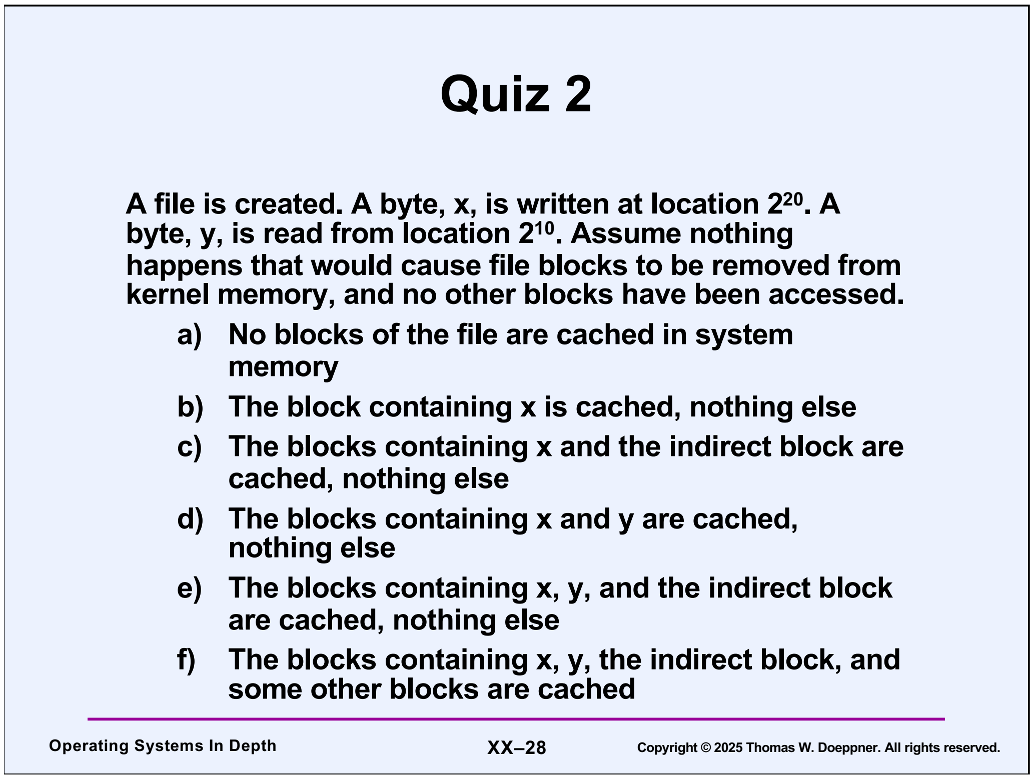
```
typedef struct fs {
  char fs_dev[STR_MAX];
  char fs_mountpt[STR_MAX];
  struct vnode *fs_vnodecovered;
  struct vnode *fs_root;
  fs_ops_t *fs_op;
      /* function pointers */
  void *fs_i;
      /* extra stuff in subclasses */
} fs_t;
```

```
typedef struct s5fs {
  blockdev_t *s5f_bdev;
     // refers to fs's mobj
  s5_super_t s5f_super;
  kmutex_t s5f_mutex;
  fs_t *s5f_fs;
} s5fs_t;
```

The data structure on the left, which we've seen before, is the VFS representation of a file system. Its fs_i pointer, for s5fs file systems, points to the data structure on the right, representing the s5fs-specific information about a file system. (Its s5f_fs pointer points back to the fs_t.) The s5f_bdev member refers to device-specific information about the disk area on which the file system resides. Included there is a mobj_t that's used to cache blocks from the file system.

Weenix Address Space Rep


vmmap_t
vmarea_t 0–7fff x, shared
vmarea_t 8000–1afff rw, private
vmarea_t 1b000–1bfff rw, private
vmarea_t 7fffd000–7fffffff rw, private
mobj_t
vnode_t
file
mobj_t
vnode_t
mobj_t
s5fs_t
pframe_t
pframe_t
pframe_t
pframe_t
mobj_t
vnode_t

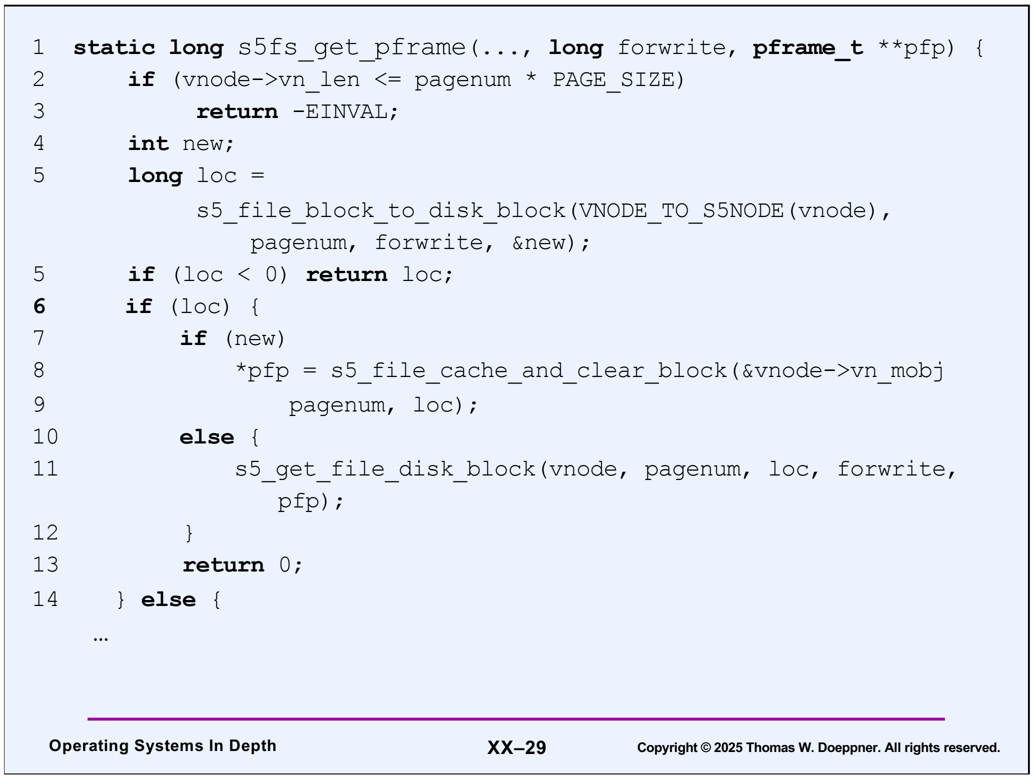


Quiz 2

A file is created. A byte, x, is written at location 2^{20}. A byte, y, is read from location 2^{10}. Assume nothing happens that would cause file blocks to be removed from kernel memory, and no other blocks have been accessed.

a) No blocks of the file are cached in system memory
b) The block containing x is cached, nothing else
c) The blocks containing x and the indirect block are cached, nothing else
d) The blocks containing x and y are cached, nothing else
e) The blocks containing x, y, and the indirect block are cached, nothing else
f) The blocks containing x, y, the indirect block, and some other blocks are cached

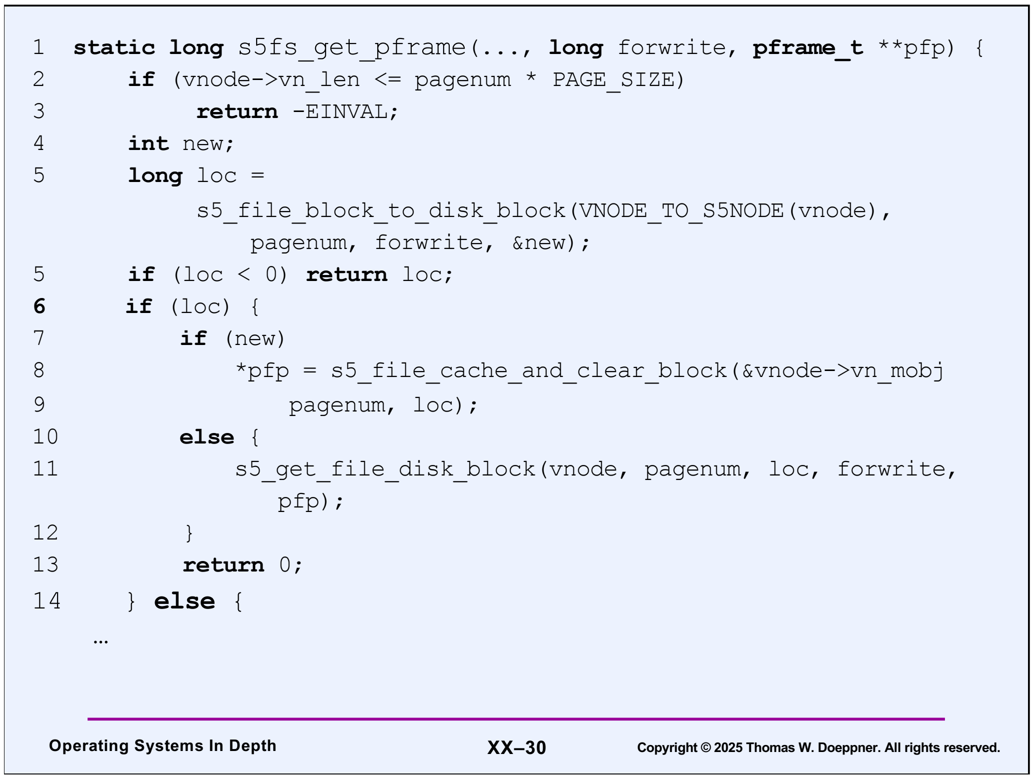
In this problem we're concerned only about blocks of the file. For example, we aren't concerned about caching of blocks related to the directory the file appears in.



```
1  static long s5fs_get_pframe(..., long forwrite, pframe_t **pfp) {
2      if (vnode->vn_len <= pagenum * PAGE_SIZE)
3           return -EINVAL;
4      int new;
5      long loc =
            s5_file_block_to_disk_block(VNODE_TO_S5NODE(vnode),
                pagenum, forwrite, &new);
5      if (loc < 0) return loc;
6      if (loc) {
7          if (new)
8              *pfp = s5_file_cache_and_clear_block(&vnode->vn_mobj
9                  pagenum, loc);
10         else {
11             s5_get_file_disk_block(vnode, pagenum, loc, forwrite,
                   pfp);
12         }
13         return 0;
14     } else {
     …
```

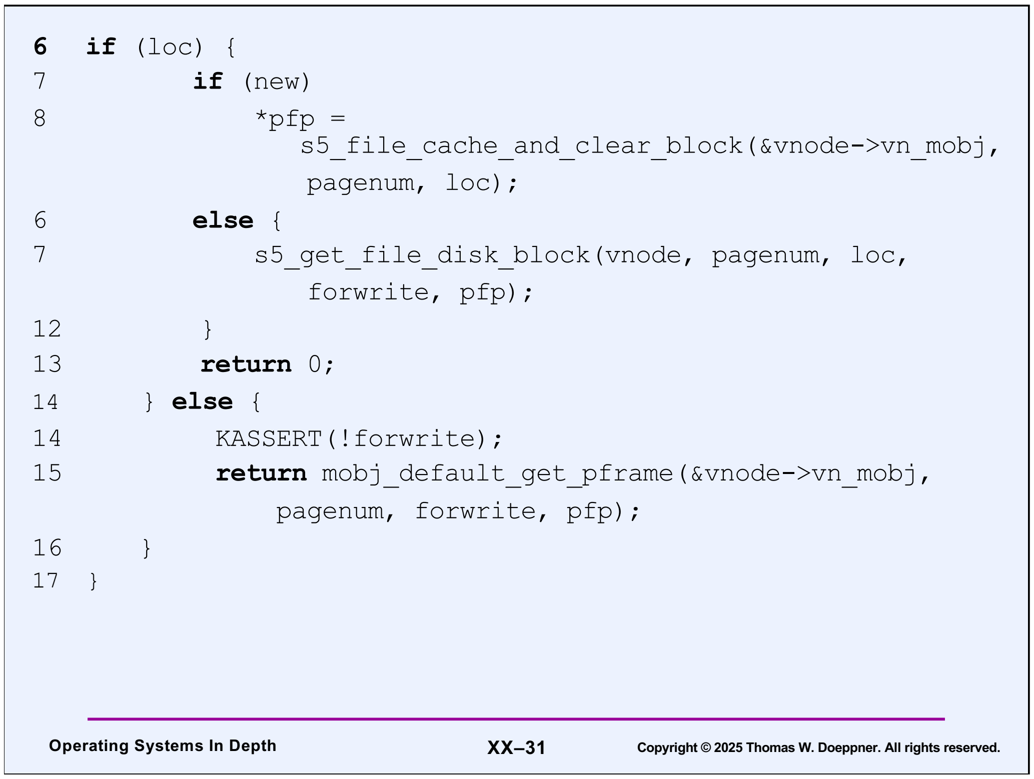
This is the code, from the stencil, for **s5fs_get_pframe** (it's in kernel/fs/s5fs/s5fs.c). This is called for a particular page (block) number of a file to get it into kernel memory and return (via the **pfp** argument) the **pframe** struct that refers to that page. If the block is a sparse block and the intent is merely to read the block, a zero is returned. But if the **forwrite** flag is set, then the reason for calling **s5fs_get_pframe** is to write to that block, and, in this case, we need to allocate space for it on disk.

At line 4, **s5_file_block_to_disk_block** is called to translate (via the disk map) from the offset of the block within the file to the offset of the block within the file system (disk). (The disk map is found in the inode, which is obtained from the vnode by the macro **VNODE_TO_S5NODE**.) If the result of the translation is negative, there was an error. If the result is positive, it's the block number on disk. If the result is 0, then it's not on disk because it's a sparse block.



```
1  static long s5fs_get_pframe(..., long forwrite, pframe_t **pfp) {
2      if (vnode->vn_len <= pagenum * PAGE_SIZE)
3           return -EINVAL;
4      int new;
5      long loc =
            s5_file_block_to_disk_block(VNODE_TO_S5NODE(vnode),
                pagenum, forwrite, &new);
5      if (loc < 0) return loc;
6      if (loc) {
7          if (new)
8              *pfp = s5_file_cache_and_clear_block(&vnode->vn_mobj
9                  pagenum, loc);
10         else {
11             s5_get_file_disk_block(vnode, pagenum, loc, forwrite,
                   pfp);
12         }
13         return 0;
14     } else {
   …
```

If the block is on disk, we will read it from disk. However, if the block is to be modified (**forwrite** was set), then **s5_file_block_to_disk_block** allocated space on disk for the block and adjusted the disk map to refer to it, and returned that (positive) block number. But if the block had been read previously, it would be in the file's mobj. At line 7 if a new block was allocated on disk, then, at line 8, it is cleared and inserted into the mobj cache. If it was an existing block,, at line 11 it is read from disk and the address of the pframe_t referencing the page is returned via pfp.



```
6   if (loc) {
7          if (new)
8              *pfp =
                  s5_file_cache_and_clear_block(&vnode->vn_mobj,
                  pagenum, loc);
6          else {
7              s5_get_file_disk_block(vnode, pagenum, loc,
                  forwrite, pfp);
12         }
13         return 0;
14      } else {
14          KASSERT(!forwrite);
15          return mobj_default_get_pframe(&vnode->vn_mobj,
                pagenum, forwrite, pfp);
16      }
17  }
```

If the value returned by **s5_file_block_to_disk_block** is 0, then the desired block is not on disk because it's a sparse block, and control goes to line 14. Note that if **forwrite** had been set, then a block would have been allocated by **s5_file_block_to_disk_block** and **loc** would have been positive. Thus, if we get here, **forwrite** should be 0. The call to **mobj_default_get_pframe** allocates a page frame for the block, fills it with zeroes, and adds it to the file's mobj.

Quiz 3

Suppose a thread does a *read* system call (which calls *s5fs_get_pframe*) to read a portion of a block that is sparse. It then writes data to the block, using the *write* system call. Will, as part of handling this write, *mobj_default_get_pframe* be called?

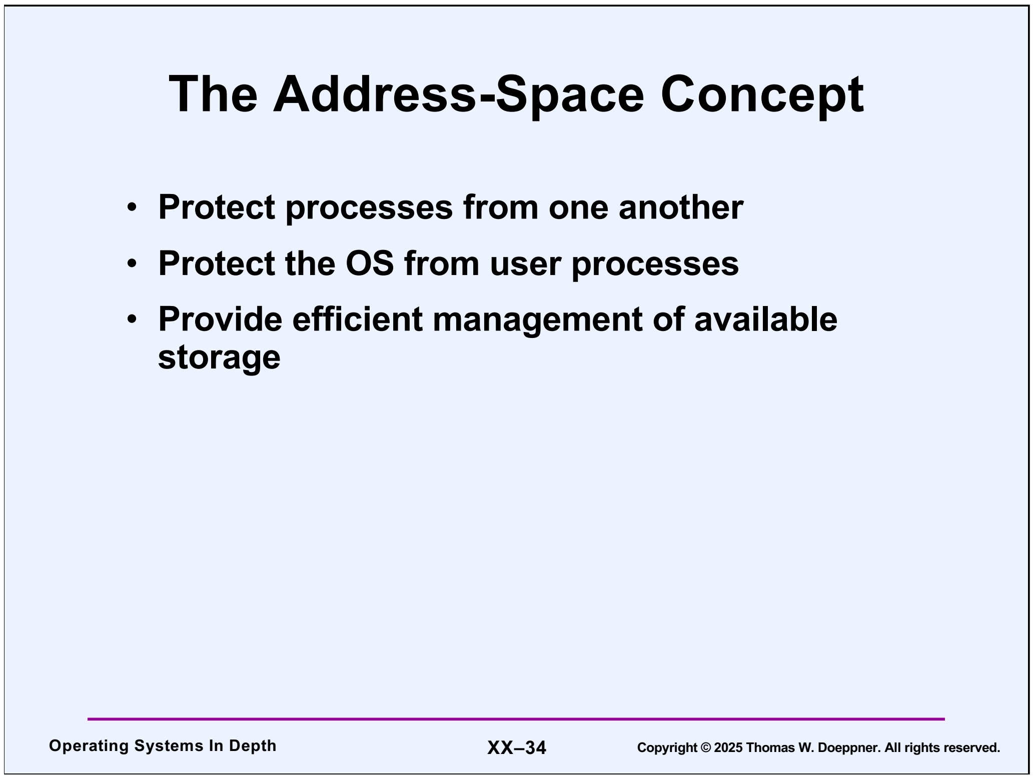
a) yes, since the block is sparse, *mobj_default_get_pframe* must be called to zero the block, then modify a portion of it

b) yes, for some other reason

c) no, the block was zeroed by the *read* call

d) no, the block doesn't need to be zeroed and the caller of *s5fs_get_pframe* will fill it in

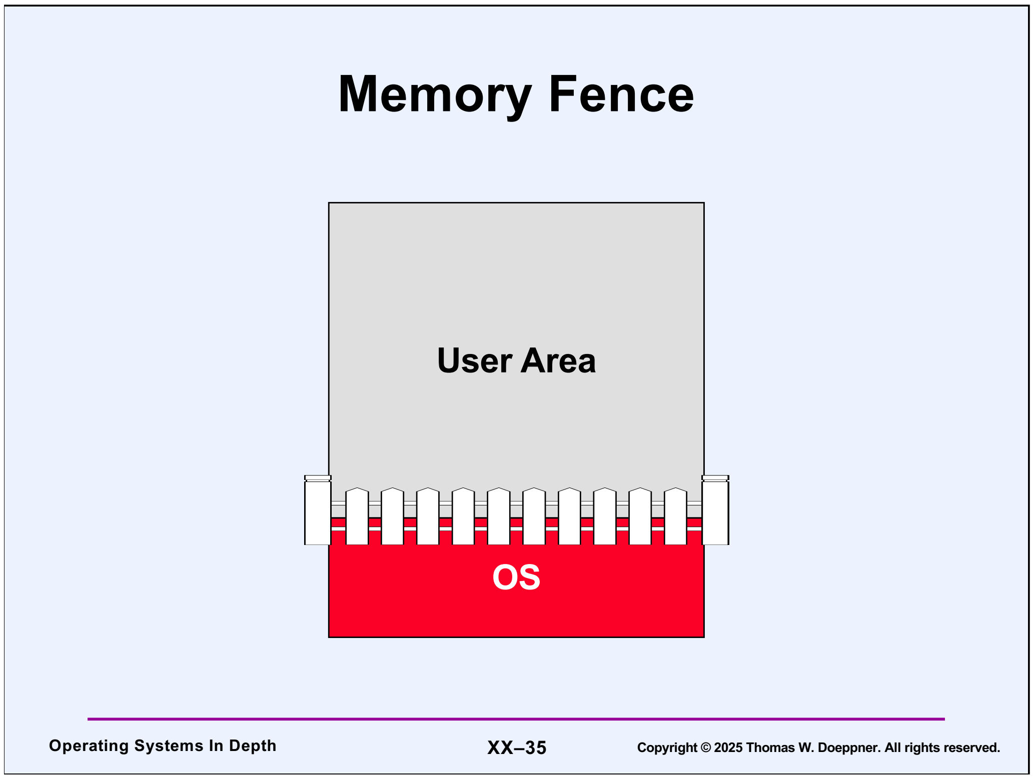
Memory Management Part 1



The Address-Space Concept

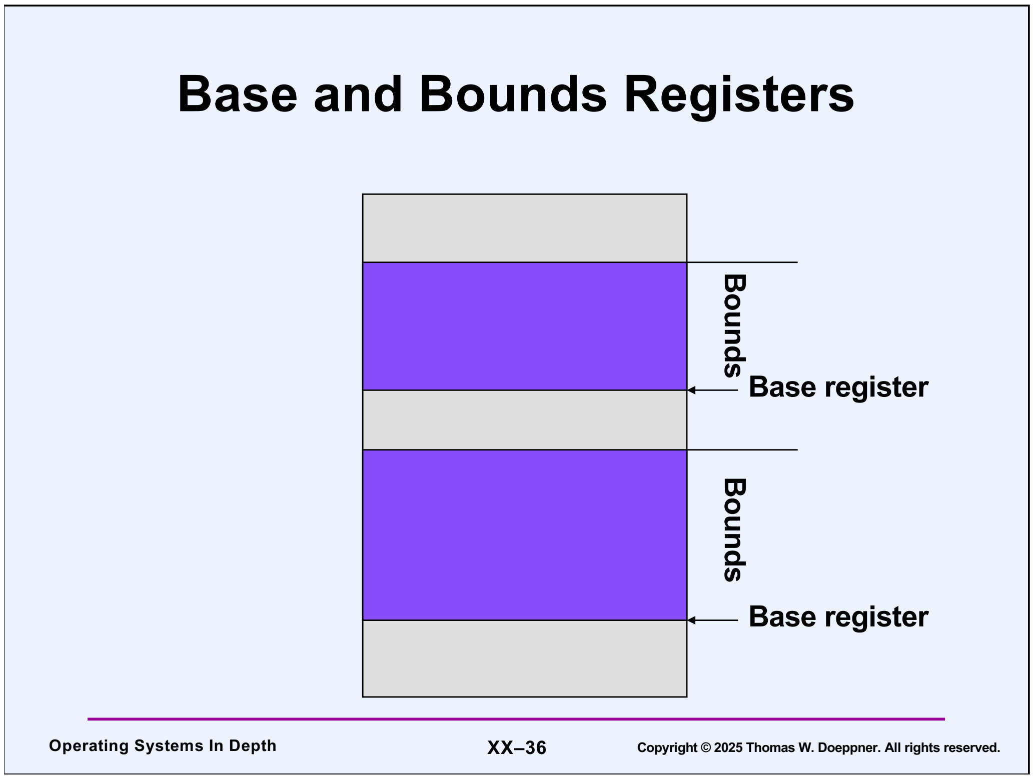
- Protect processes from one another
- Protect the OS from user processes
- Provide efficient management of available storage

The concept of the address space is fundamental in most of today's operating systems. Threads of control executing in different address spaces are protected from one another, since none of them can reference the memory of any of the others. In most systems (such as Unix), the operating system resides in address space that is shared with all processes, but protection is employed so that user threads cannot access the operating system. What is crucial in the implementation of the address-space concept is the efficient management of the underlying primary and secondary storage.

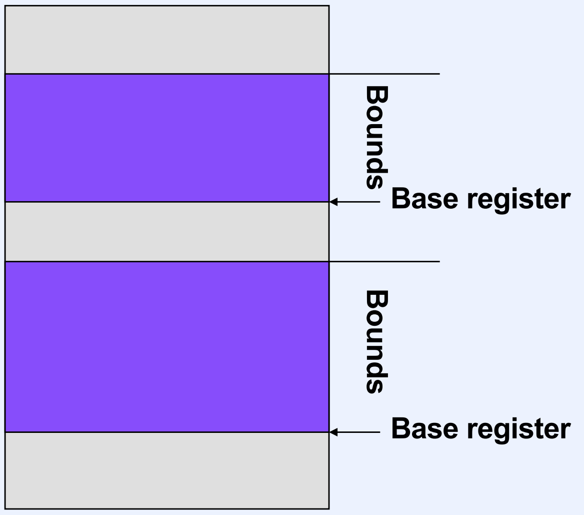


Memory Fence

Early approaches to managing the address space were concerned primarily with protecting the operating system from the user. One technique was the hardware-supported concept of the **memory fence**: an address was established below which no user mode access was allowed. The operating system was placed below this point in memory and was thus protected from the user.



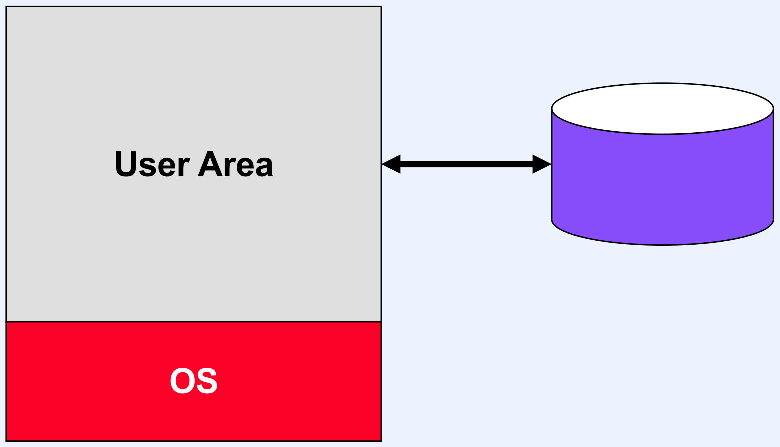
Base and Bounds Registers



The memory-fence approach protected the operating system, but did not protect user processes from one another. (This wasn't an issue for many systems—there was only one user process at a time.) Another technique, still employed in some of today's systems, is the use of **base and bounds registers** to restrict a process's memory references to a certain range. Each address generated by a user process was first compared with the value in the bounds register to make certain that it did not reference a location beyond the process's range of memory, and then was modified by adding to it the value in the base register, ensuring that it did not reference a location before the process's range of memory.

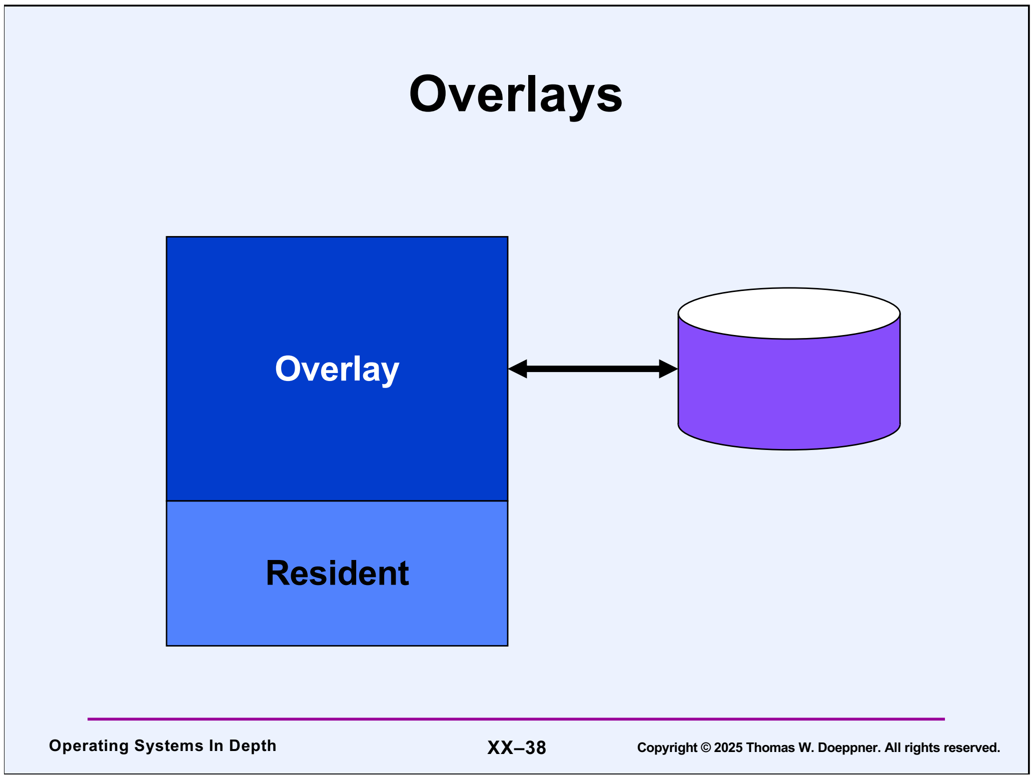
A further advantage of this technique was to ensure that a process would be loaded into what appeared to be location 0—thus no relocation was required at load time.

Swapping

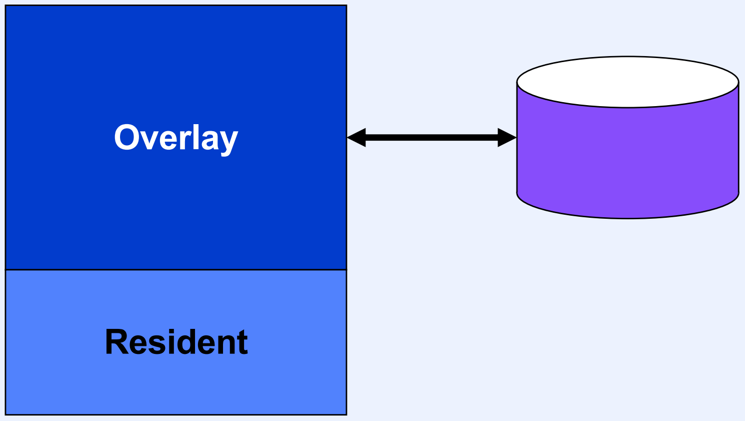


Swapping is a technique, still in use today, in which the images of entire processes are transferred back and forth between primary and secondary storage. An early use of it was for (slow) time-sharing systems: when a user paused to think, his or her process was swapped out and that of another user was swapped in. This allowed multiple users to share a system that employed only the memory fence for protection.

Base and bounds registers made it feasible to have a number of processes in primary memory at once. However, if one of these processes was inactive, swapping allowed the system to swap this process out and swap another process in. Note that the use of the base register is very important here: without base registers, after a process is swapped out, it would have to be swapped into the same location in which it resided previously.

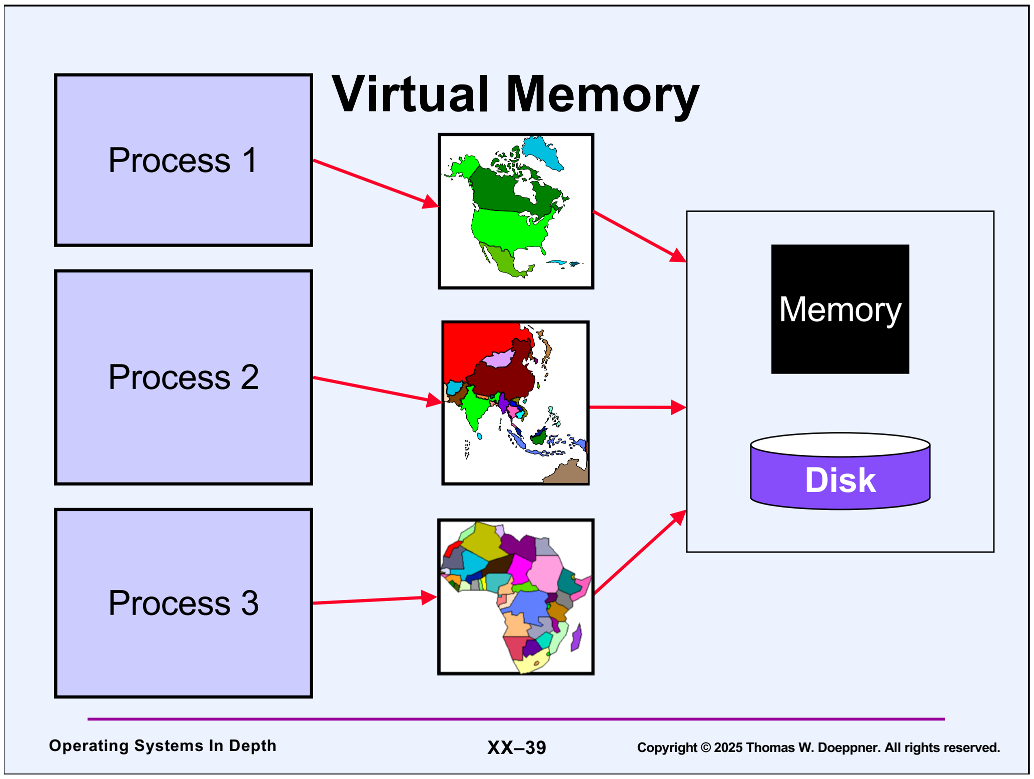


Overlays



The concept of overlays is similar to the concept of swapping, except that it applies to pieces of images rather than whole images and the user is in charge. Say we have 100 kilobytes of available memory and a 200-kilobyte program. Clearly, not all the program can be in memory at once. The user might decide that one portion of the program should always be resident, while other portions of the program need be resident only for brief periods. The program might start with routines A and B loaded into memory. A calls B; B returns. Now A wants to call C, so it first reads C into the memory previously occupied by B (it **overlays** B), and then calls C. C might then want to call D and E, though there is only room for one at a time. So, C first calls D, D returns, then C overlays D with E and then calls E.

The advantage of this technique is that the programmer has complete control of the use of memory and can make the necessary optimization decisions. The disadvantage is that the programmer must make the necessary decisions to make full use of memory (the operating system doesn't help out). Few programmers can make such decisions wisely, and fewer still want to try.

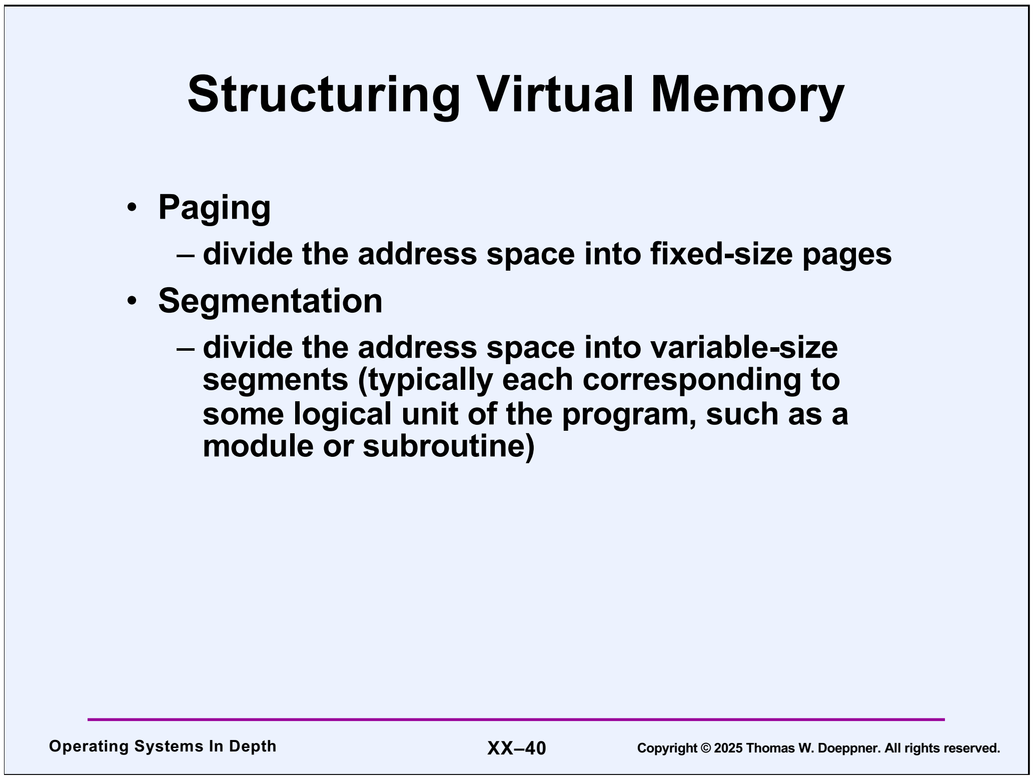


Virtual Memory

One way to look at virtual memory is as an automatic overlay technique: processes "see" an address space that is larger than the amount of real memory available to them; the operating system is responsible for the overlaying.

Put more abstractly (and accurately), virtual memory is the support of an address space that is independent of the size of primary storage. Some sort of mapping technique must be employed to map virtual addresses to primary and secondary stores. In the typical scenario, the computer hardware maps some virtual addresses to primary storage. If a reference is made to an unmapped address, then a fault occurs (a **page fault**) and the operating system is called upon to deal with it. The operating system might then find the desired virtual locations on secondary storage and transfer them to primary storage. Or the operating system might decide that the reference is illegal and deliver an addressing exception to the process.

As with base and bounds registers, the virtual memory concept allows us to handle multiple processes simultaneously, with the processes protected from one another.



Structuring Virtual Memory

- Paging
  - divide the address space into fixed-size pages
- Segmentation
  - divide the address space into variable-size segments (typically each corresponding to some logical unit of the program, such as a module or subroutine)

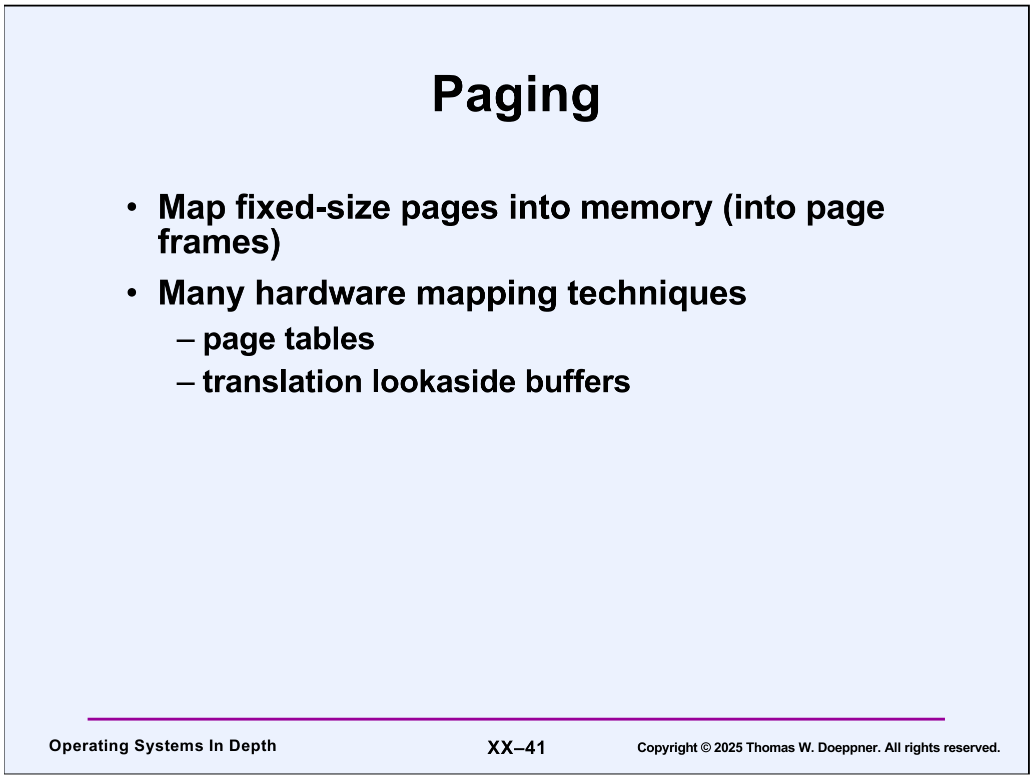
There are two basic approaches to structuring virtual memory: it is divided either into fixed-size **pages** or into variable-size **segments**.

With the former approach, the management of available storage is simplified, since memory is always allocated one page at a time. However, there is some waste due to internal fragmentation—the typical program does not require an integral number of pages, but, on the average, requires memory whose size is 1/2 page less than an integral multiple of the page size.

With the latter approach, memory allocation is more difficult, since allocations are for varying amounts of memory. This may lead to external fragmentation, in which memory is wasted (as we saw in discussing dynamic storage allocation) because there are a number of free areas of memory too small to be of any use. The advantage of segmentation is that it is a useful organizational tool—programs are composed of segments, each of which can be dealt with (e.g. fetched by the operating system) independently of the others.

Segment-based schemes were popular in the ’60s and ’70s but are less so today, primarily because the advantages of segmentation do not outweigh the extra costs due to complexity of the hardware and software used to manage it. We will restrict our discussion to page-based schemes.

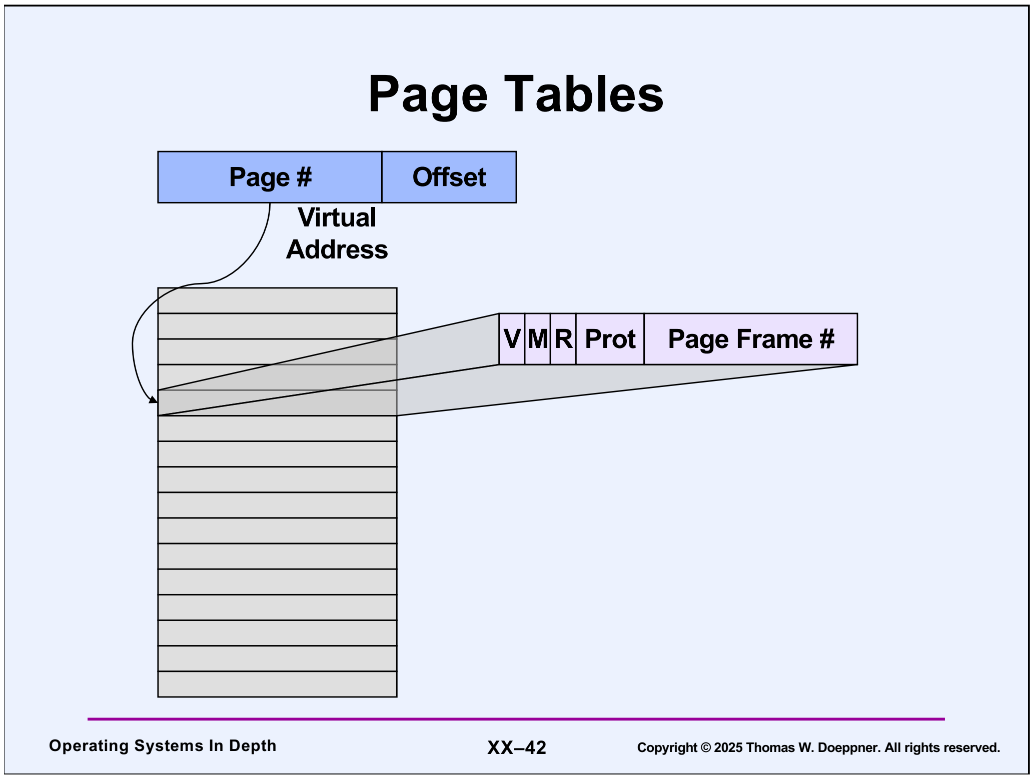
There is also a compromise approach, **paged segmentation** (as opposed to **segmented paging**, which we discuss later), in which each segment is divided up into pages. This approach serves to make segmentation a more viable alternative, but doesn’t serve well enough.



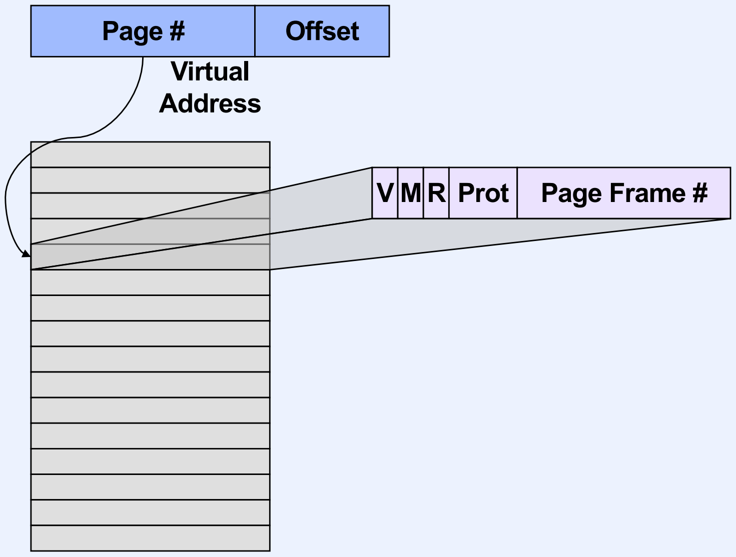
Paging

- Map fixed-size pages into memory (into page frames)
- Many hardware mapping techniques
  - page tables
  - translation lookaside buffers

There are a number of hardware-based facilities for the support of paging. The most common of them is the use of **page tables**, which are tables implementing complete maps from virtual memory to real memory. In the following pages we examine a number of variations on page-table design. Another approach, either used alone or in conjunction with the other two, is the use of caches called **translation lookaside buffers** (TLBs).

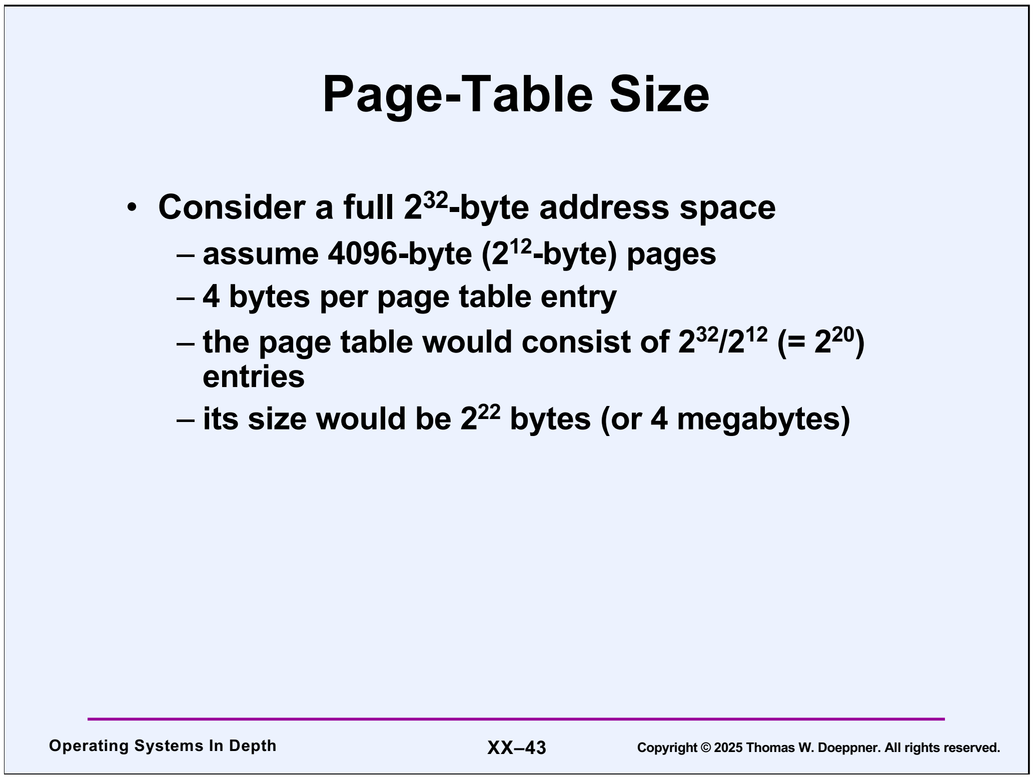


Page Tables



A page table is an array of **page table entries**. Suppose we have a 32-bit virtual address and a page size of 4096 bytes. The 32-bit address might be split into two parts: a 20-bit **page number** and a 12-bit **offset** within the page. When a thread generates an address, the hardware uses the page-number portion as an index into the page-table array to select a page-table entry, as shown in the picture. If the page is in primary storage (i.e. the translation is valid), then the validity bit in the page-table entry is set, and the page-frame-number portion of the page-table entry is the high-order bits of the location in primary memory where the page resides. (Primary memory is thought of as being subdivided into pieces called **page frames**, each exactly big enough to hold a page; the address of each of these page frames is at a “page boundary,” so that its low-order bits are zeros.) The hardware then appends the offset from the original virtual address to the page-frame number to form the final, real address.

If the **validity bit** of the selected page-table entry is zero, then a page fault occurs, and the operating system takes over. Other bits in a typical page-table entry include a **reference bit**, which is set by the hardware whenever the page is referenced, and a **modified bit**, which is set whenever the page is modified. We will see how these bits are used later. The **page-protection bits** indicate who is allowed access to the page and what sort of access is allowed. For example, the page can be restricted for use only by the operating system, or a page containing executable code can be write-protected, meaning that read accesses are allowed but not write accesses.

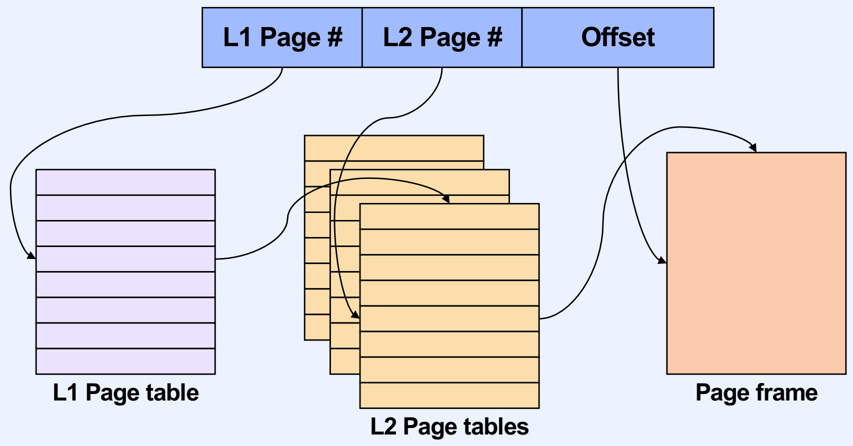


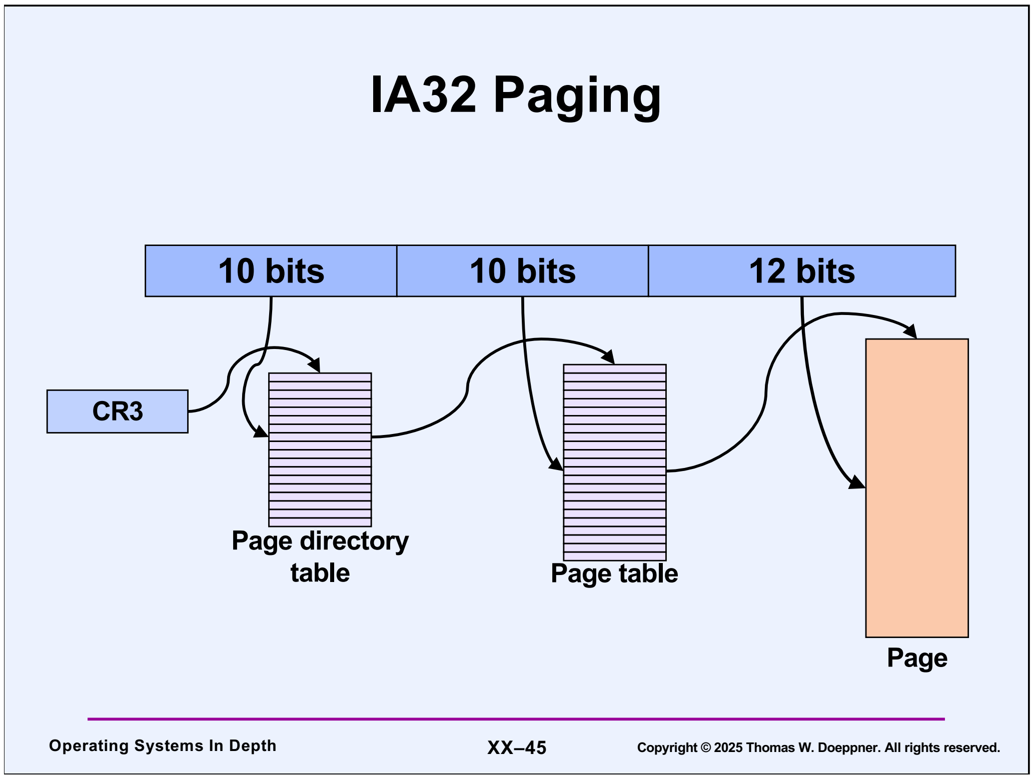
Page-Table Size

- Consider a full 2^{32}-byte address space
  - assume 4096-byte (2^{12}-byte) pages
  - 4 bytes per page table entry
  - the page table would consist of $2^{32}/2^{12}$ (= 2^{20}) entries
  - its size would be 2^{22} bytes (or 4 megabytes)

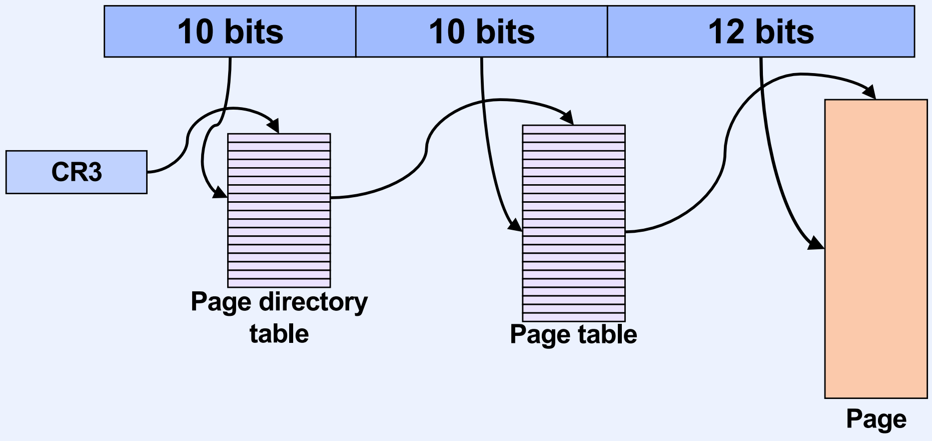
Not too long ago, 4 megabytes was considered to be a lot of memory!

Forward-Mapped Page Table

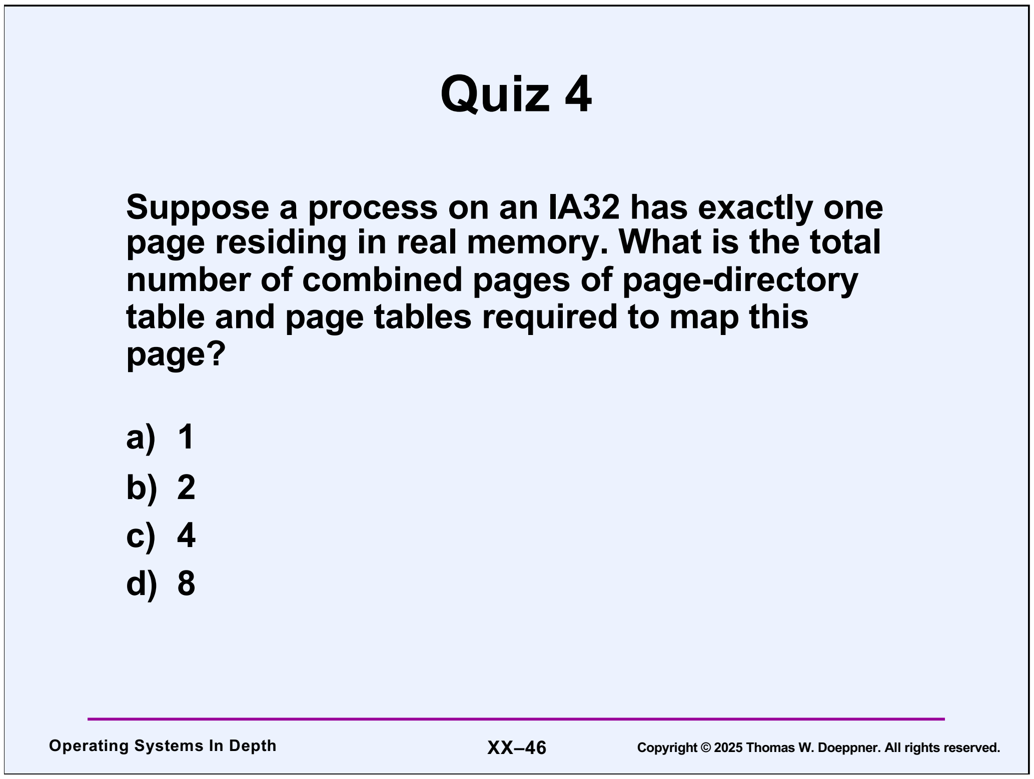




IA32 Paging



The IA32 architecture employs a two-level page table providing a means for reducing the memory requirements of the address map. The high-order 10 bits of the 32-bit virtual address are an index into what's called the page directory table. Each of its entries refer to a page table, whose entries are indexed by the next 10 bits of the virtual address. Its entries refer to individual pages; the offset within the page is indexed by the low-order 12 bits of the virtual address. The current page directory is pointed to by a special register known as CR3 (control register 3), whose contents may be modified only in privileged mode. The page directory must reside in real memory when the address space is in use, but it is relatively small (1024 4-byte entries: it's exactly one page in length). Though there are potentially a large number of page tables, only those needed to satisfy current references must be in memory at once.



Quiz 4

Suppose a process on an IA32 has exactly one page residing in real memory. What is the total number of combined pages of page-directory table and page tables required to map this page?

a) 1
b) 2
c) 4
d) 8

Note we're not counting the page that's mapped, only the page-directory and page table used to map it.