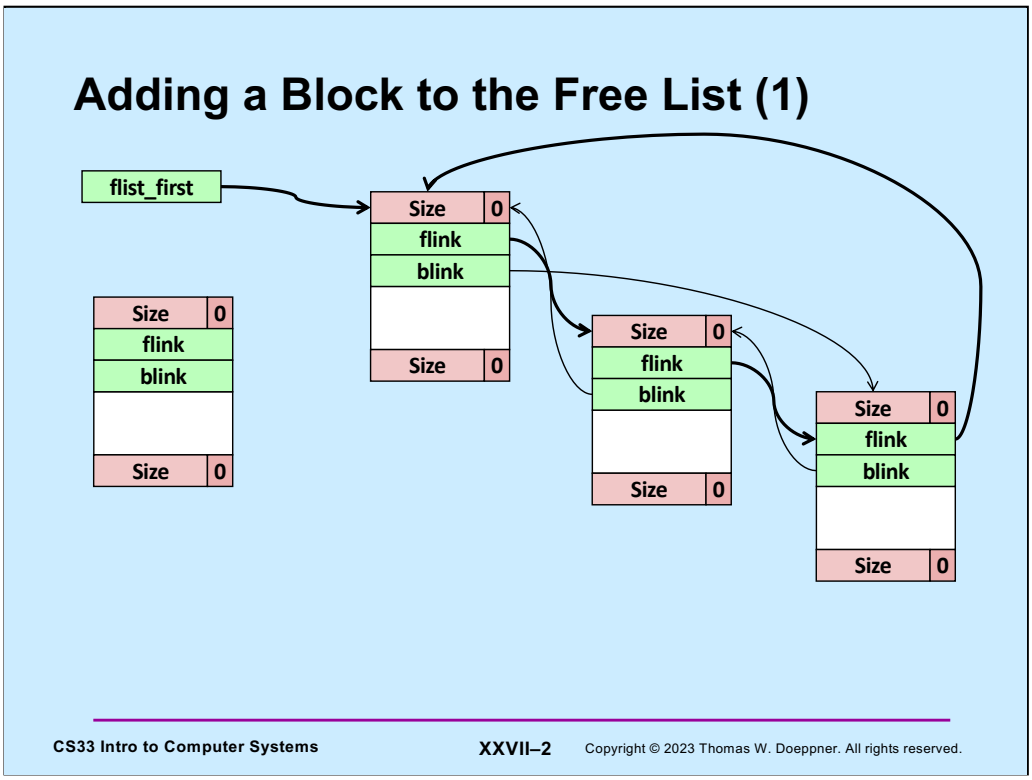
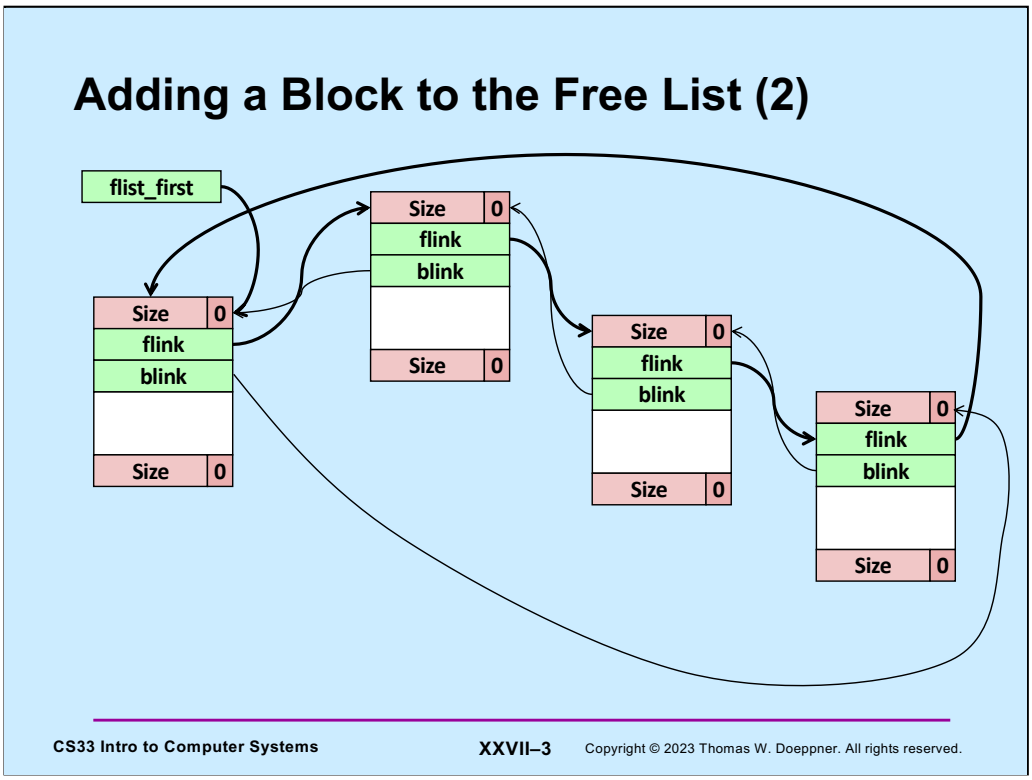
CS 33

Storage Allocation (2)



An important operation is to add a block to the beginning of the free list. We start with a picture of the free list.



Here's what it looks like after we add the block.

Accessing the Object

```
block_t *block_flink(block_t *b) {
  return (block_t *)b->payload[0];
}

void block_set_flink(block_t *b, block_t *next) {
  b->payload[0] = (size_t)next;
}

block_t *block_blink(block_t *b) {
  return (block_t *)b->payload[1];
}

void block_set_blink(block_t *b, block_t *next) {
  b->payload[1] = (size_t)next;
}
```


Here are a few more simple functions we need to access and set fields of blocks.

Insertion Code

```
void insert_free_block(block_t *fb) {
  assert(!block_allocated(fb));
  if (flist_first != NULL) {
    block_t *last =
      block_blink(flist_first);
    block_set_flink(fb, flist_first);
    block_set_blink(fb, last);
    block_set_flink(last, fb);
    block_set_blink(flist_first, fb);
  } else {
    block_set_flink(fb, fb);
    block_set_blink(fb, fb);
  }
  flist_first = fb;
}
```


Using our functions, here's the code to insert a block at the beginning of the free list.

Performance

- Won't all the calls to the accessor functions slow things down a lot?
  - yes — not just a lot, but tons
- Why not use macros (#define) instead?
  - the textbook does this
  - it makes the code impossible to debug
    - » gdb shows only the name of the macro, not its body
- What to do????

We've used a lot of functions without thinking about their effect on performance. While we know that the overhead of a function call is not great, there is still some overhead that might best be eliminated.

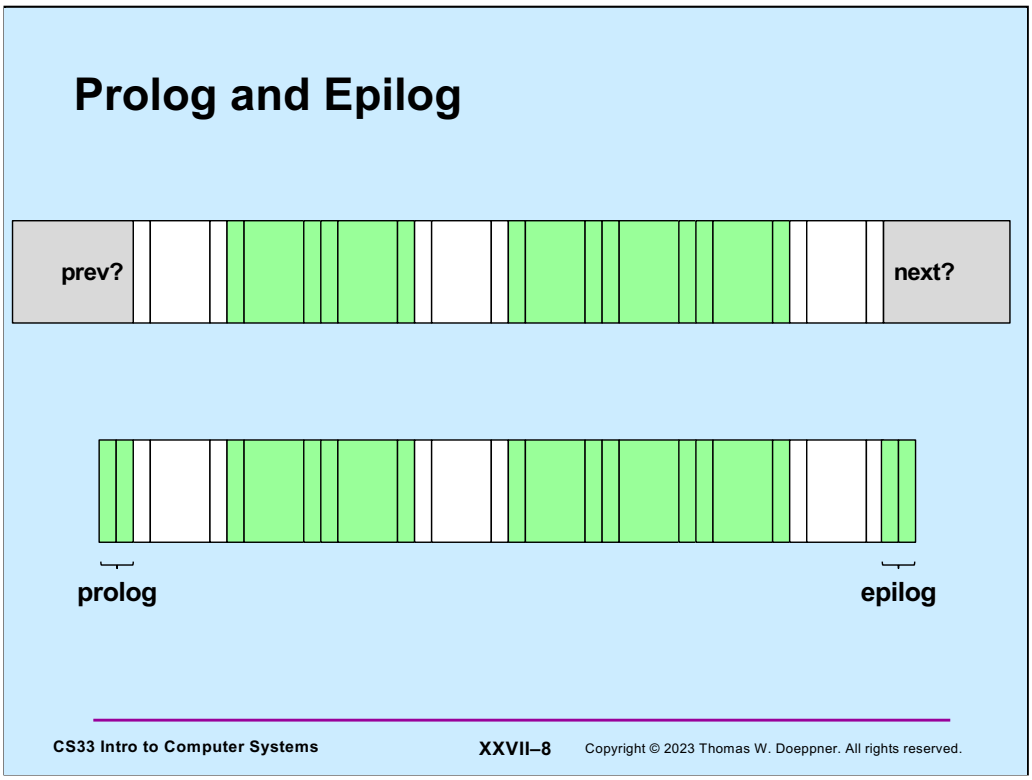
Inline Functions

```
static inline size_t block_size(
    block_t *b) {
  return b->size & -2;
}
```

- when debugging (–O0), the code is implemented as a normal function
  - » easy to debug with gdb
- when optimized (–O1, –O2), calls to the function are replaced with the body of the function
  - » no function-call overhead

If we declare a function to be **inline**, the C compiler is instructed to replace calls to the function with its actual code (unless –O0 is specified). Thus, inlined functions have no function-call overhead (though they do increase the total size of our code, which does come at some cost).

Note that inline functions are declared to be *static*. This makes it possible to have two .c files that use an inline function, with one compiled with –O0 and the other perhaps with –O1 – since the function is static, it can be different in the two source files.



The slide shows our heap, with allocated blocks shown in green and free blocks in white. At either end of each block are its tags.

An issue that comes up when implementing malloc/free is dealing with the first and last blocks, whether they are allocated or free. What is the **prev** block relative to the first block? What is the **next** block relative to the last block? Having to special-case the first and last blocks can help make your code unnecessarily complicated. To avoid these complications, we use **prolog** and **epilog** blocks. These are blocks of minimum size (containing just two tags and no payload) that are marked **allocated**. They are on either end of the list. Since they're marked allocated, when a check is made of the **prev** block relative to the first real block, it will always appear to be allocated, and similarly with the **next** block relative to the last real block.

Thus, the initial heap might consist of three blocks: the prolog, a block representing the initial free space, and an epilog. Of course, when the heap is expanded by calling **sbrk**, the epilog must be moved to the new end of the heap.

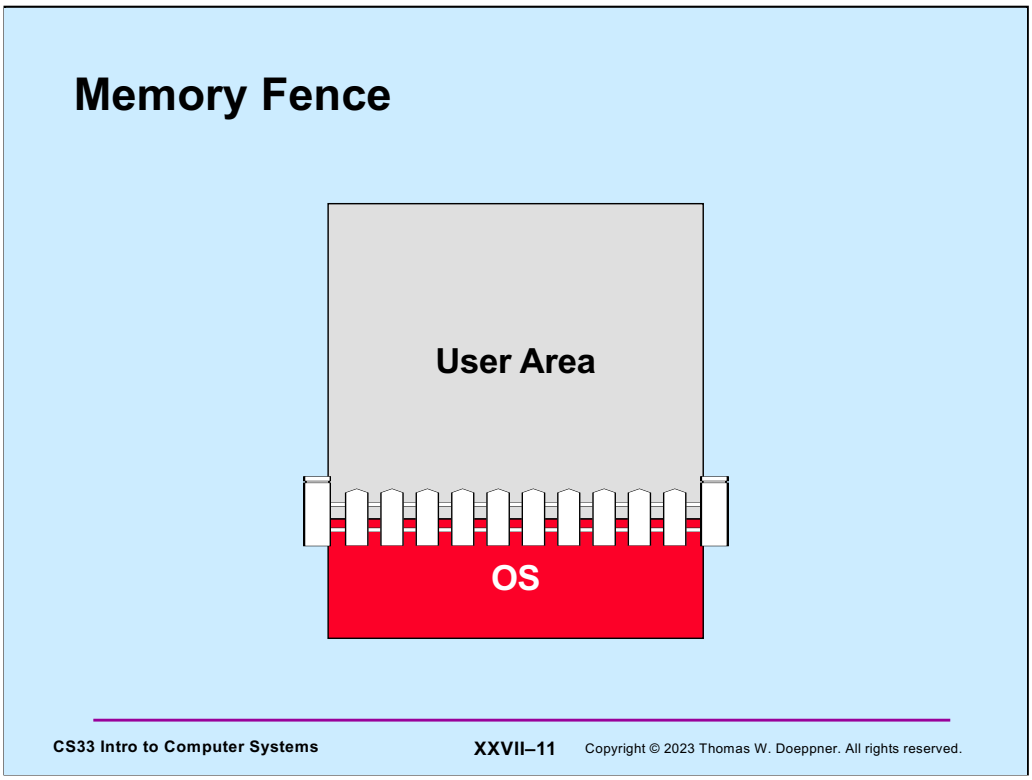
CS 33

Virtual Memory

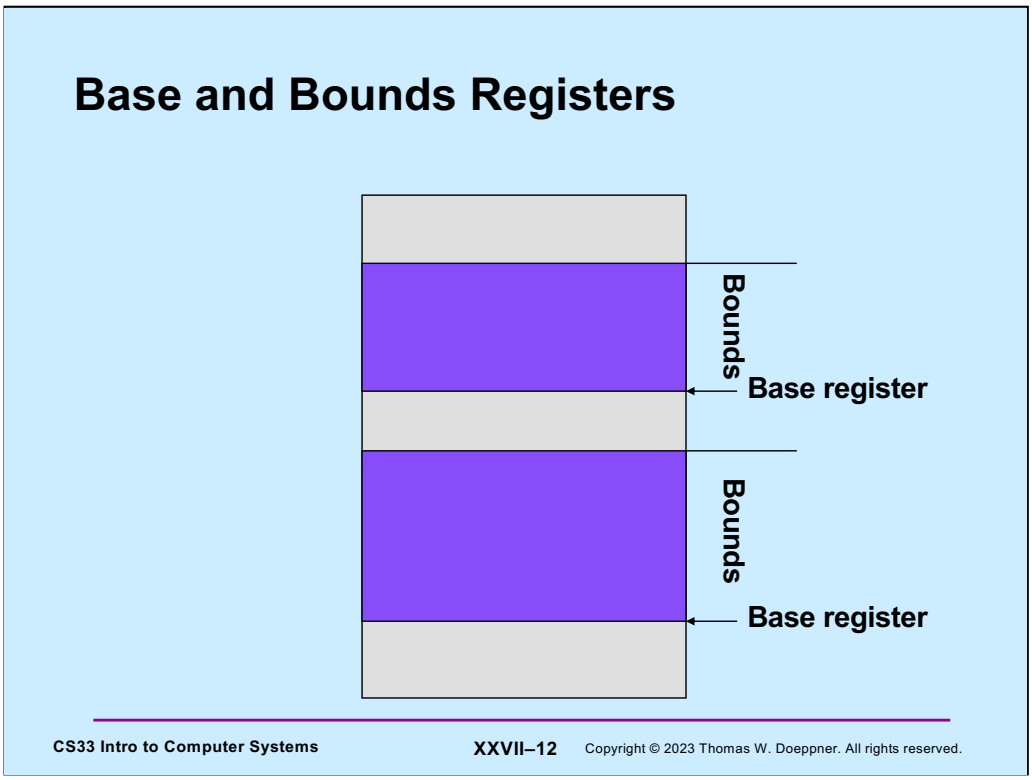
The Address-Space Concept

- **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.

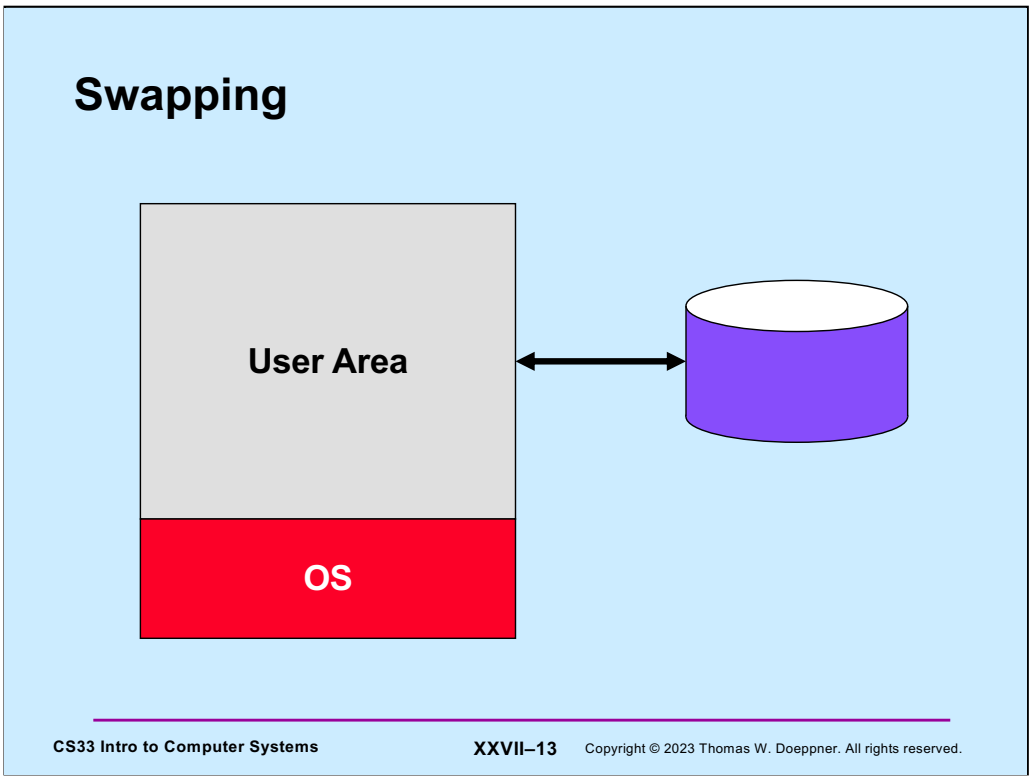


Early approaches to managing the address space were concerned primarily with protecting the operating system from the user. One technique was the hardware-supported concept of the **memory fence**: an address was established below which no user mode access was allowed. The operating system was placed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



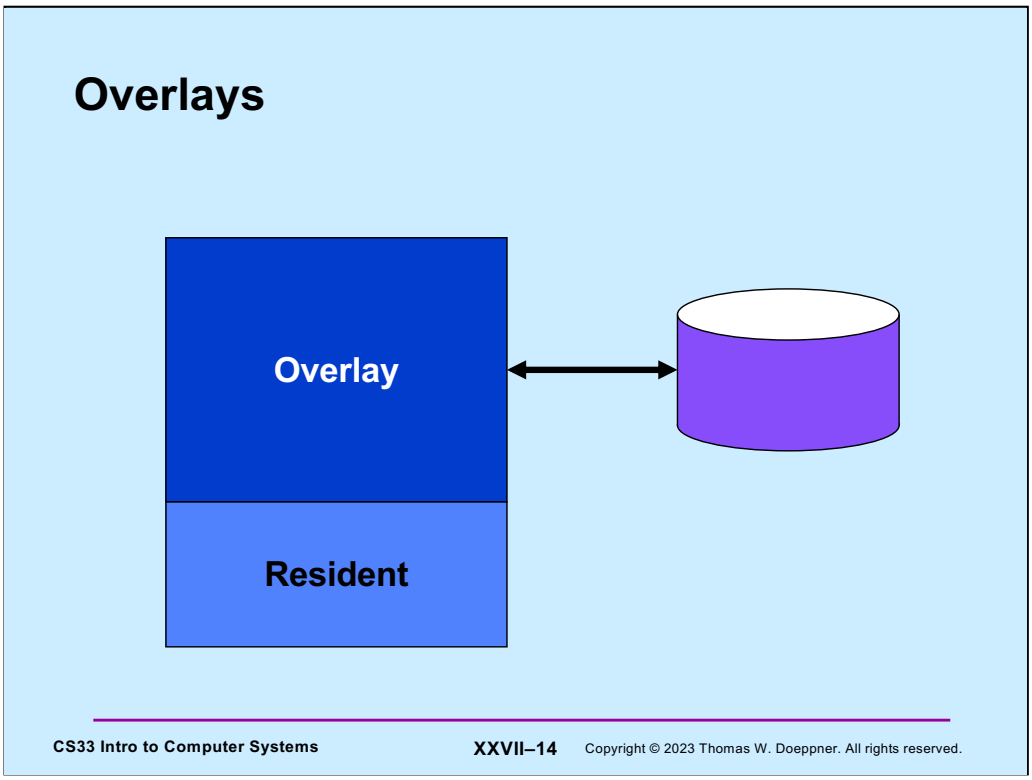
The memory-fence approach protected the operating system, but did not protect user processes from one another. (This wasn't an issue for many systems—there was only one user process at a time.) Another technique, still employed in some of today's systems, is the use of **base and bounds registers** to 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.



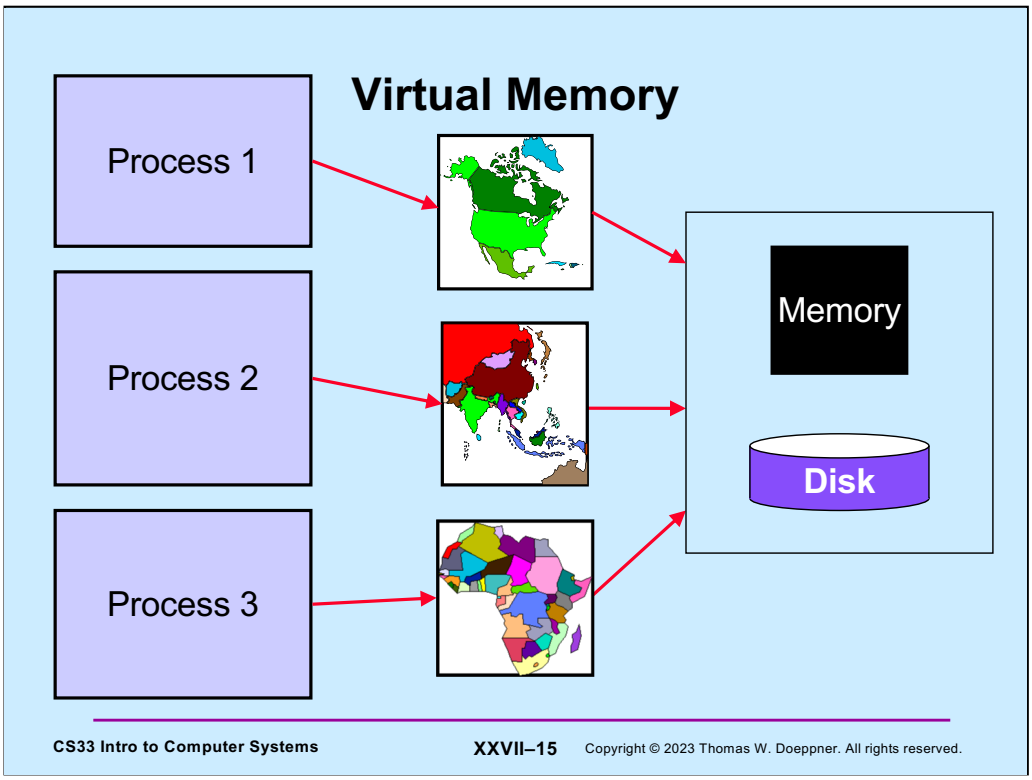
Swapping is a technique, still in use today, in which the images of entire processes are transferred back and forth between primary and secondary storage. An early use of it was for (slow) time-sharing systems: when a user paused to think, his or her process was swapped out and that of another user was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



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.

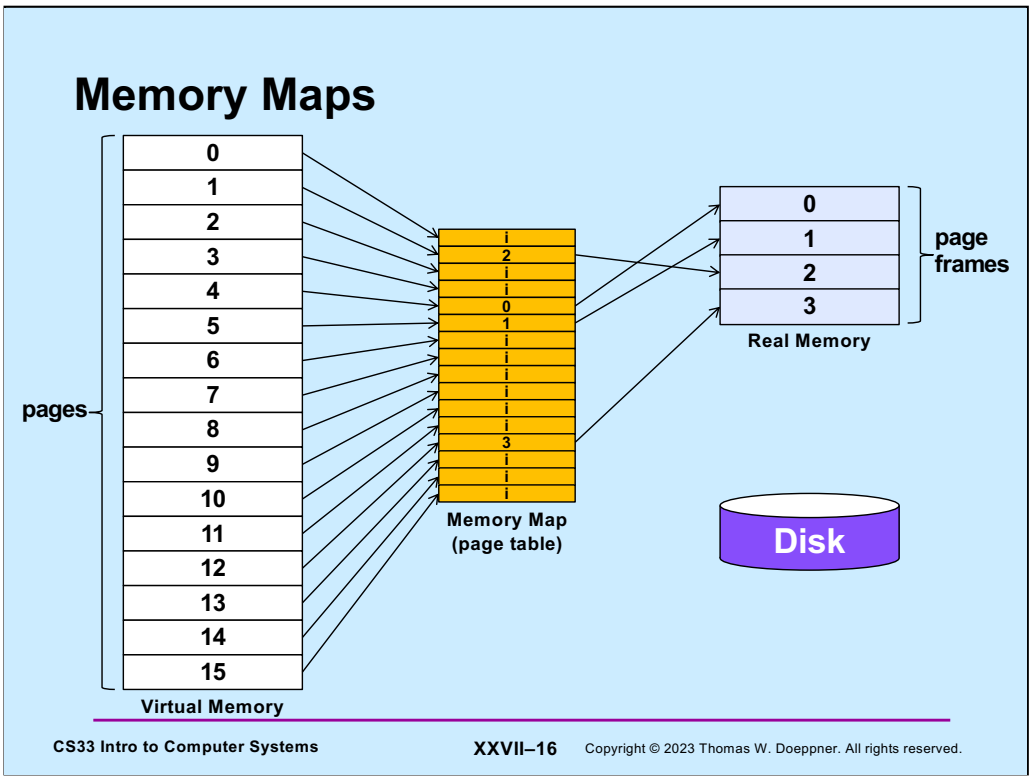
The advantage of this technique is that the programmer has complete control of the use of memory and can make the necessary optimization decisions. The disadvantage is that the programmer **must** make the necessary decisions to make full use of memory (the operating system doesn't help out). Few programmers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



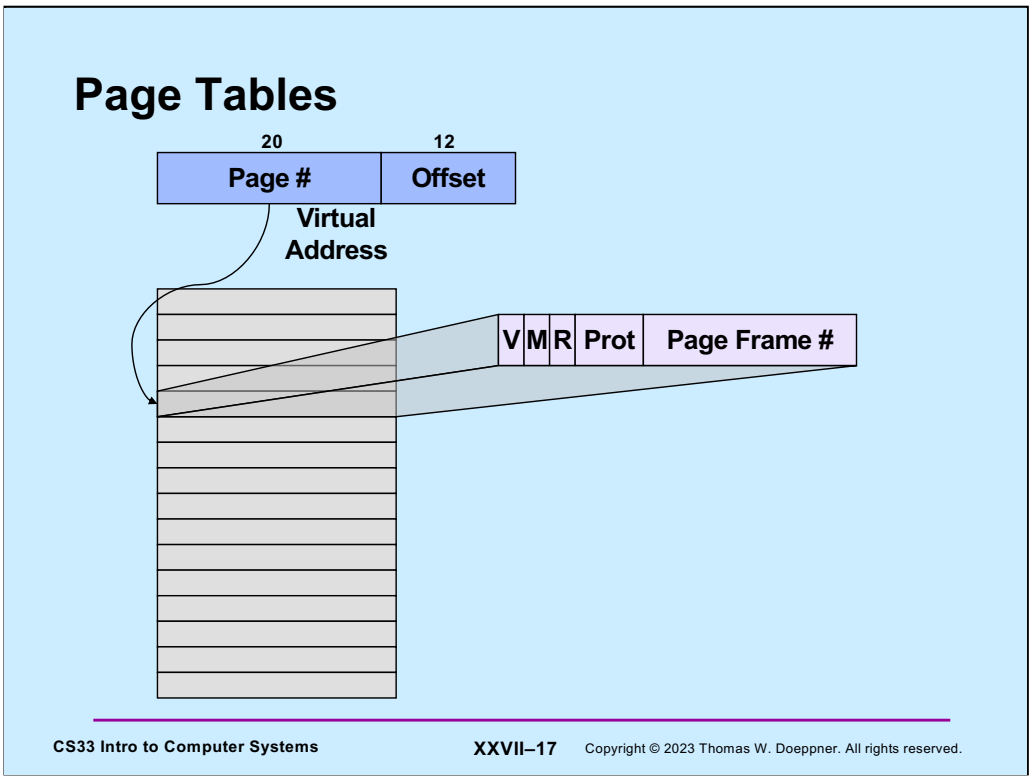
One way to look at virtual memory is as an automatic overlay technique: processes "see" an address space that is larger than the amount of real memory available to them; the operating system is responsible for the overlaying.

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 (such as a disk) and transfer them to primary storage. Or the operating system might decide that the reference is illegal and deliver a seg fault 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.



Virtual memory (what the program sees) is divided into fixed-size pages (on the x86 these are usually 4 kilobytes in size). Real memory (DRAM) is also divided into fixed-size pieces, called page frames (though they're often referred to simply as pages). A memory map, implemented in hardware and often called a page table, translates references to virtual-memory pages into references to real-memory page frames. In general, virtual memory is larger than real memory, thus not all pages can be mapped to page frames. Those that are not are said to have invalid translations.



A page table is an array of *page table entries*. Suppose we have, as is the usual case for the x86, 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 in this lecture. 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.

Quiz 1

How many 2^{12}-byte pages fit in a 32-bit address space?

a) a little over a 1000
b) a little over a million
c) a little over a billion
d) none of the above

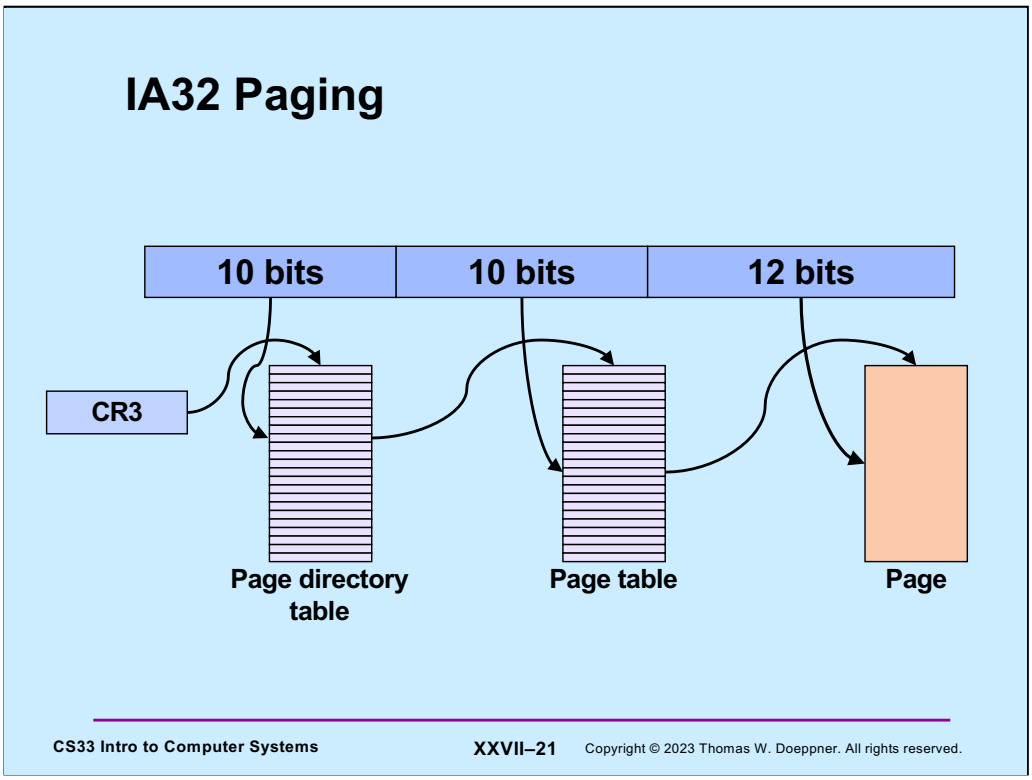
VM is Your Friend ...

- Not everything has to be in memory at once
  - pages brought in (and pushed out) when needed
  - unallocated parts of the address space consume no memory
    - » e.g., hole between stack and dynamic areas
- What's mine is not yours (and vice versa)
  - address spaces are disjoint
- Sharing is ok though ...
  - address spaces don't have to be disjoint
    - » a single page frame may be mapped into multiple processes
- I don't trust you (or me)
  - access to individual pages can be restricted
    - » read, write, execute, or any combination

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)
    - » at $100/gigabyte
      - around $0.40
- For a 2^{64}-byte address space
  - assume 4096-byte (2^{12}-byte) pages
  - 8 bytes per page-table entry
  - the page table would consist of $2^{64}/2^{12}$ (= 2^{52}) entries
  - its size would be 2^{55} bytes (or 32 petabytes)
    - » at $1/gigabyte
      - over $33 million

In the not-all-that-distant past, 4 megabytes of memory would have cost many tens of thousands of dollars.

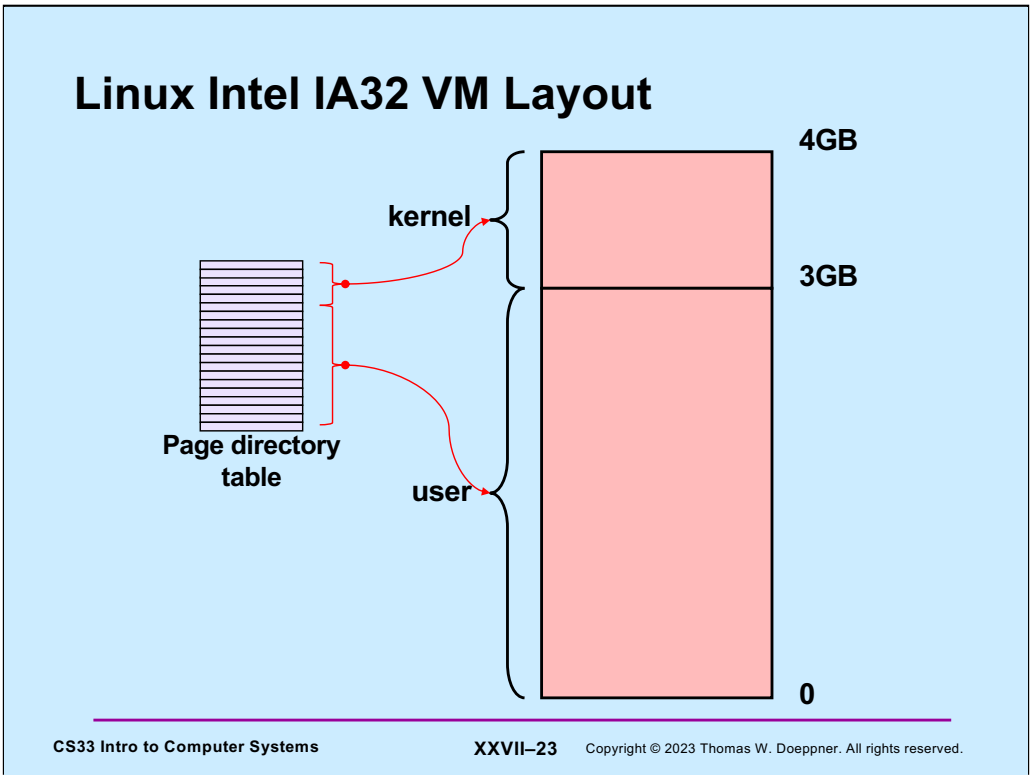


The IA32 architecture employs a two-level page table providing a means for reducing the memory requirements of the address map. The high-order 10 bits of the 32-bit virtual address are an index into what's called the page directory table. Each of its entries refer to a page table, whose entries are indexed 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 2

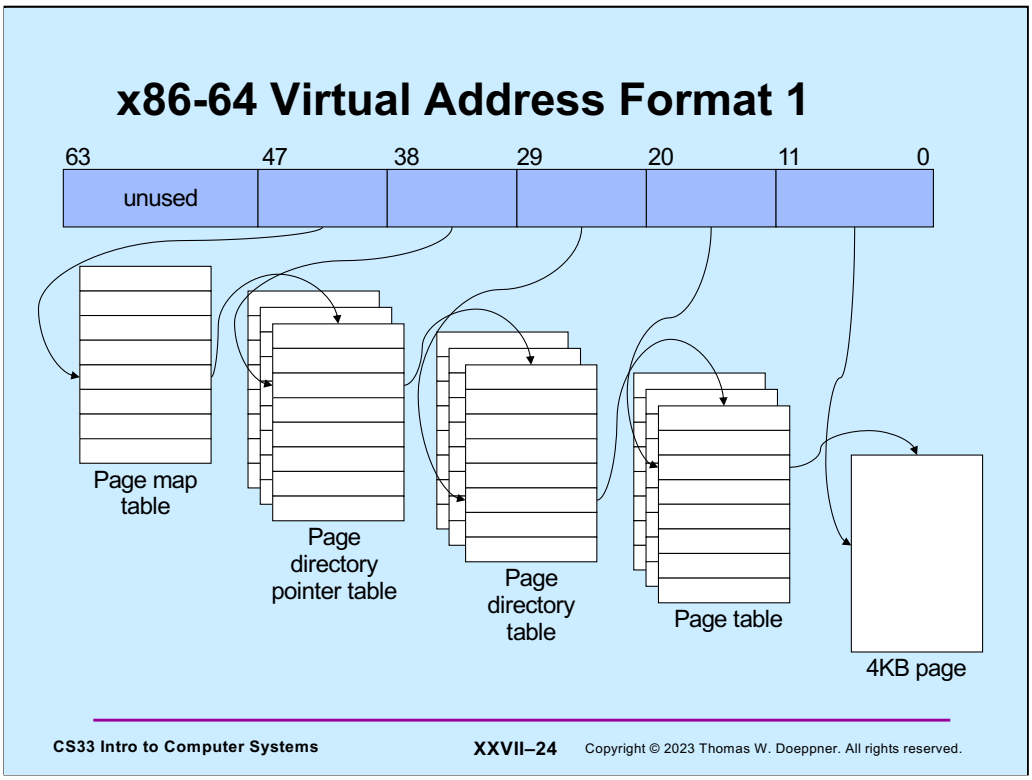
Can a page start at a virtual address that's not divisible by the page size?

a) yes
b) no

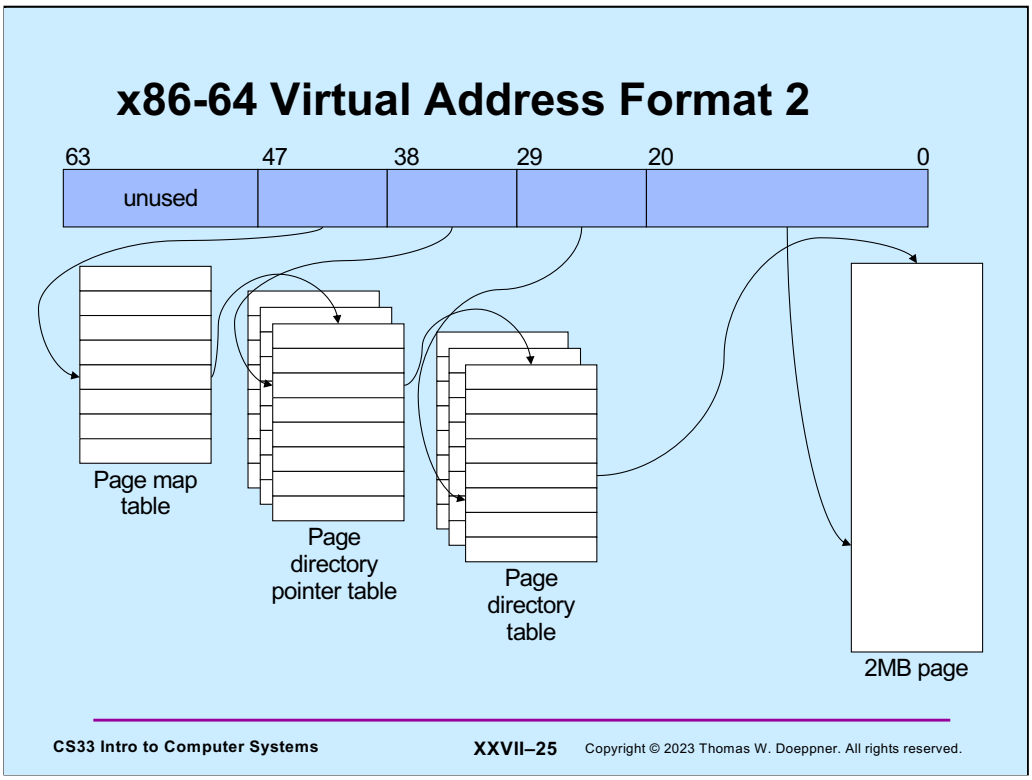


For Linux on the IA32, the OS kernel occupies the top quarter of the address space and is mapped into every process (though it may not be accessed in user mode). Each user process is mapped into the bottom three quarters of the address space — only one is mapped at a time for each processor.

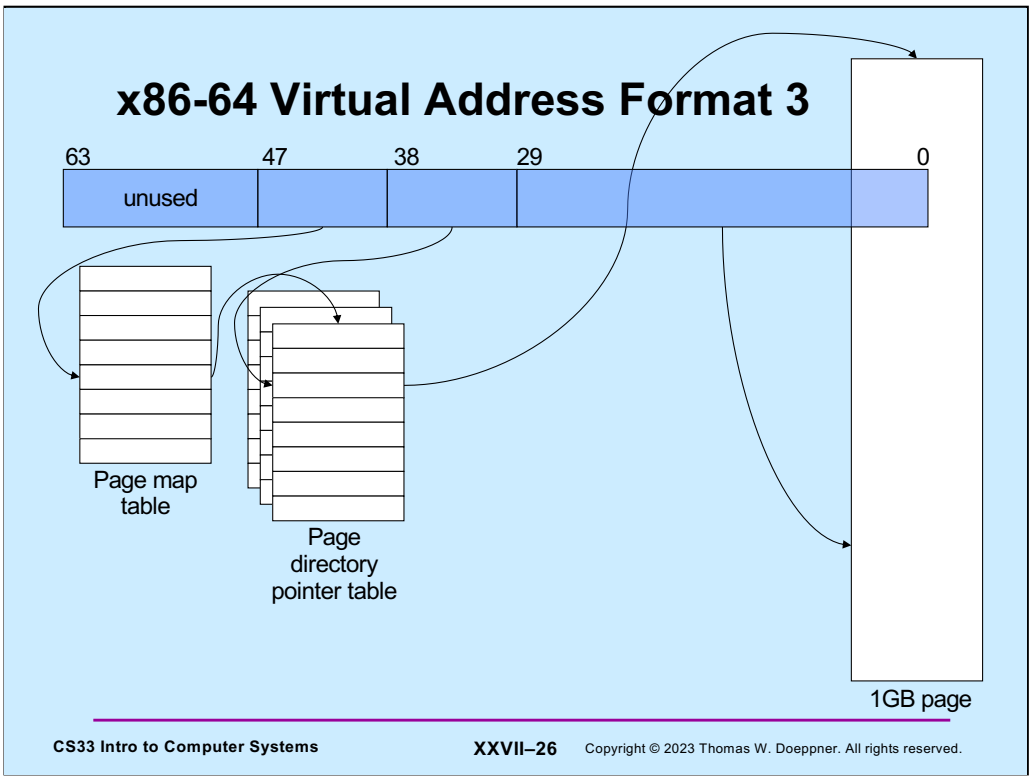
Each process has its own page-directory table describing its address space. The top quarter (256) entries are the same as for all other processes and describe the OS kernel's mappings. The bottom three quarters (768) are, in general, private to the process and describe its mappings.



For the x86-64, four levels of translation are done (the high-order 16 bits of the address are not currently used: the hardware requires that these 16 bits must all be equal to bit 47), thus it really supports “only” a 48-bit address space. Note that only the “page map table” must reside in real memory at all times. The other tables must be resident only when necessary.



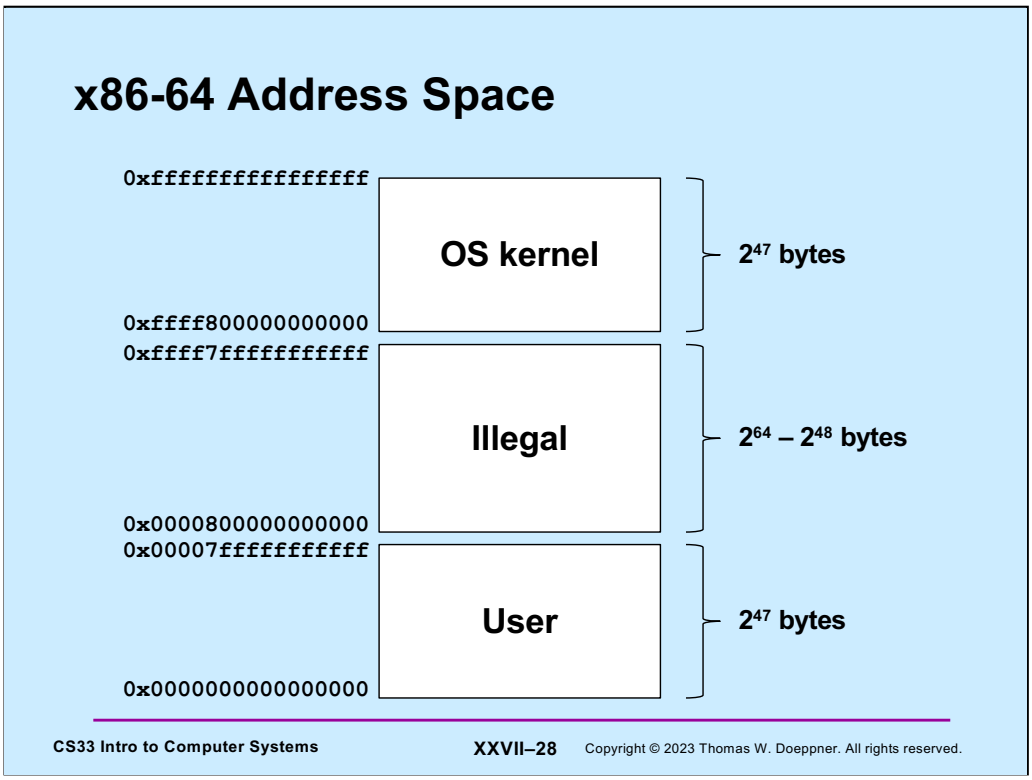
Alternatively, there may be only three levels of page tables, ending with the page-directory table an 2MB pages. Both 2MB and 4KB pages may coexist in the same address space; which is being used is indicated in the associated page-directory-table entry.



The hardware also supports 1 GB pages by eliminating the page-directory table. Not many operating systems (if any) yet take advantage of this.

Why Multiple Page Sizes?

- Fragmentation
  - for region composed of 4KB pages, average internal fragmentation is 2KB
  - for region composed of 1GB pages, average internal fragmentation is 512MB
- Page-table overhead
  - larger page sizes have fewer page tables
    - » less overhead in representing mappings



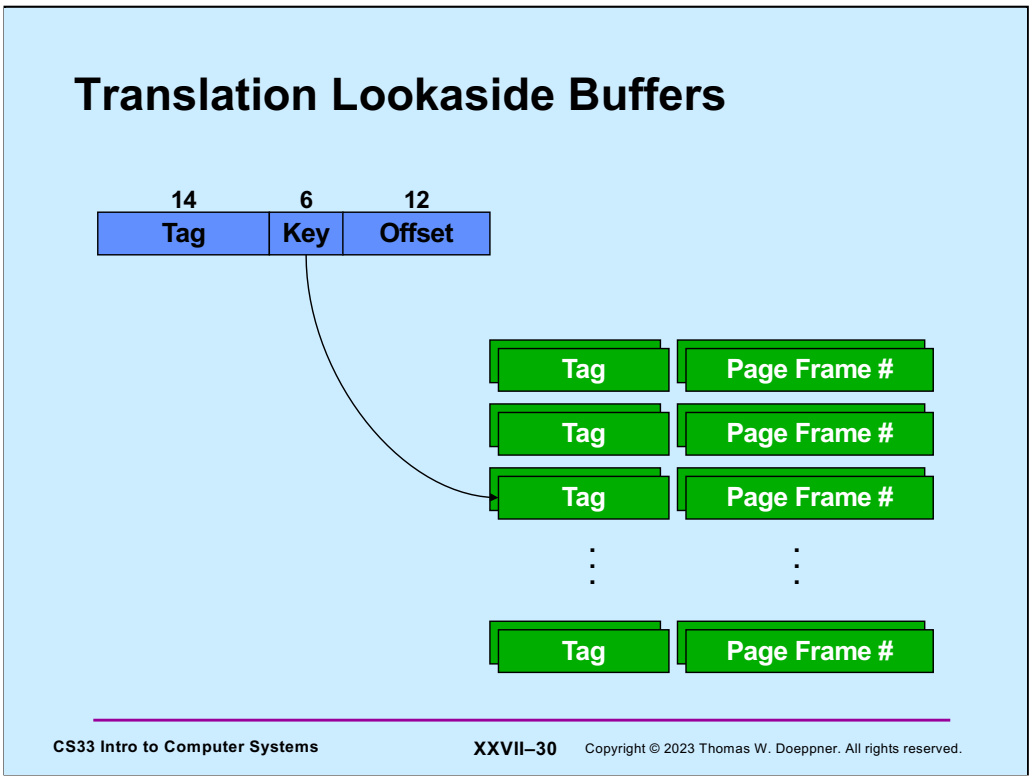
Recall that, in current implementations of the x86-64 architecture, only 48 bits of virtual address are used. Furthermore, the high-order 16 bits must be equal to bit 47. Thus the legal addresses are those at the top and at the bottom of the address space. The top addresses are used for the OS kernel, and thus mapped into all processes. The bottom address are used for each user process. The addresses in the middle (most of the address space — the slide is not drawn to scale!) are illegal and generate faults if used.

The reason for doing things this way (i.e., for the restrictions on the high-order bits) is to force the kernel to be at the top of the address space, allowing growth of the user portion as more virtual-address bits are supported.

Performance

- Page table resides in real memory (DRAM)
- A 32-bit virtual-to-real translation requires two accesses to page tables, plus the access to the ultimate real address
  - three real accesses for each virtual access
  - 3X slowdown!
- A 64-bit virtual-to-real translation requires four accesses to page tables, plus the access to the ultimate real address
  - 5X slowdown!

DRAM stands for dynamic random access memory. It's much slower (roughly by a factor of 1000) than is access to registers and caches in the processor.



To speed-up virtual-to-real translation, a special cache is maintained of recent translations — it's called the **translation lookaside buffer** (TLB). It resides in the chip, one per core and hyperthread. The TLB shown in the slide is a two-way set associative cache, which is a concept we may get to later in the course if we have time. This one assumes a 32-bit virtual address with a 4k page. Things are more complicated when multiple page sizes are supported. For example, is there just one entry for a large page that covers its entire range of addresses, or is a large page dealt with by putting into the cache multiple entries covering the large page, but each for the size of a small page? Both approaches are not only possible, but done.

Note that accessing the TLB takes far less time than accessing DRAM, so much less that the time required is negligible compared to that required to access DRAM.

Quiz 3

Recall that there is a 5x slowdown on memory references via virtual memory on the x86-64. If all references are translated via the TLB, the slowdown will be

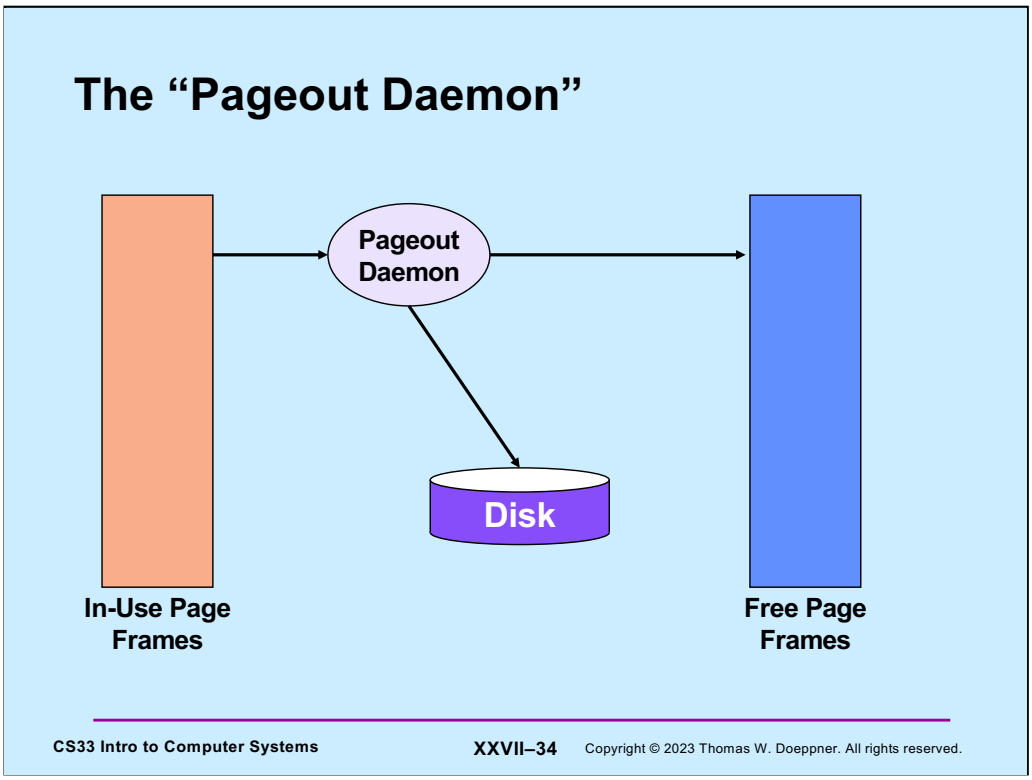
a) .5x (i.e. it will be faster, not slower)
b) 1x
c) 2x
d) 3x
e) 4x

OS Role in Virtual Memory

- Memory is like a cache
  - quick access if what's wanted is mapped via page table
  - slow if not — OS assistance required
- OS
  - make sure what's needed is mapped in
  - make sure what's no longer needed is not mapped in

Mechanism

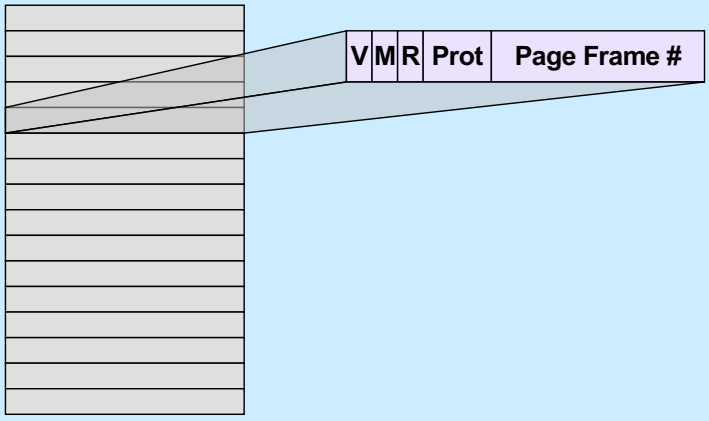
- Program references memory
  - if reference is mapped, access is quick
    - » even quicker if translation in TLB and referent in on-chip cache
  - if not, page-translation fault occurs and OS is invoked
    - » determines desired page
    - » maps it in, if legal reference



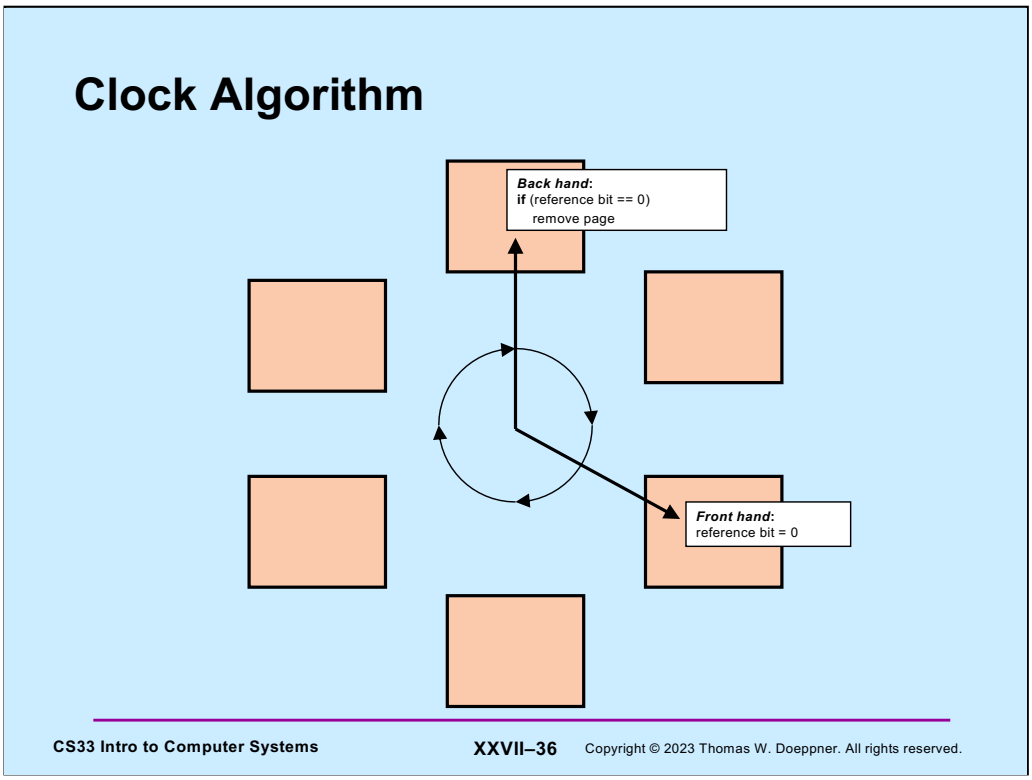
The (kernel) thread that maintains the free page-frame list is typically called the **pageout daemon**. Its job is to make certain that the free page-frame list has enough page frames on it. If the size of the list drops below some threshold, then the pageout daemon examines those page frames that are being used and selects a number of them to be freed. Before freeing a page, it must make certain that a copy of the current contents of the page exists on secondary storage. So, if the page has been modified since it was brought into primary storage (easily determined by the hardware-supported **modified bit**), it must first be written out to secondary storage. In many systems, the pageout daemon groups such pageouts into batches, so that a number of pages can be written out in a single operation, thus saving disk time. Unmodified, selected pages are transferred directly to the free page-frame list, modified pages are put there after they have been written out.

In most systems, pages in the free list get a “second chance” — if a thread in a process references such a page, there is a page fault (the page frame has been freed and could be used to hold another page), but the page-fault handler checks to see if the desired page is still in primary storage, but in the free list. If it is in the free list, it is removed and given back to the faulting process. We still suffer the overhead of a trap, but there is no wait for I/O.

Managing Page Frames



The OS can keep track of the history of page frame by use of two bits in each page-table entry: the *modify* bit, which is set by hardware whenever the associated page frame is modified, and the *referenced* bit, which is set by hardware whenever the associated page is accessed (via either a load or a store).

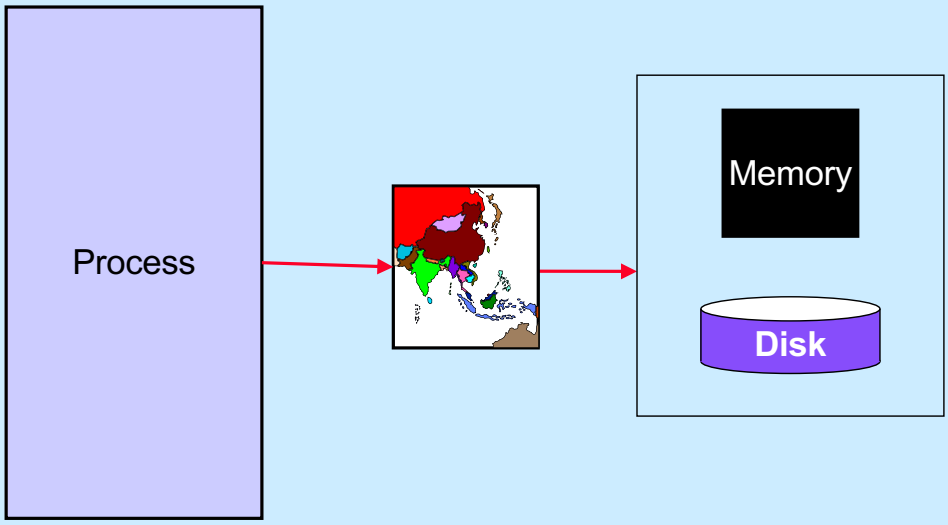


A common approach for determining which page frames are not in use is known as the clock algorithm. All active page frames are conceptually arranged in a circularly linked list. The page-out thread slowly traverses the list. The "one-handed" version of the clock algorithm, each time it encounters a page, checks the reference bit in the corresponding translation entry: if the bit is set, it clears it. If the bit is clear, it adds the page to the free list (writing it back to secondary storage first, if necessary).

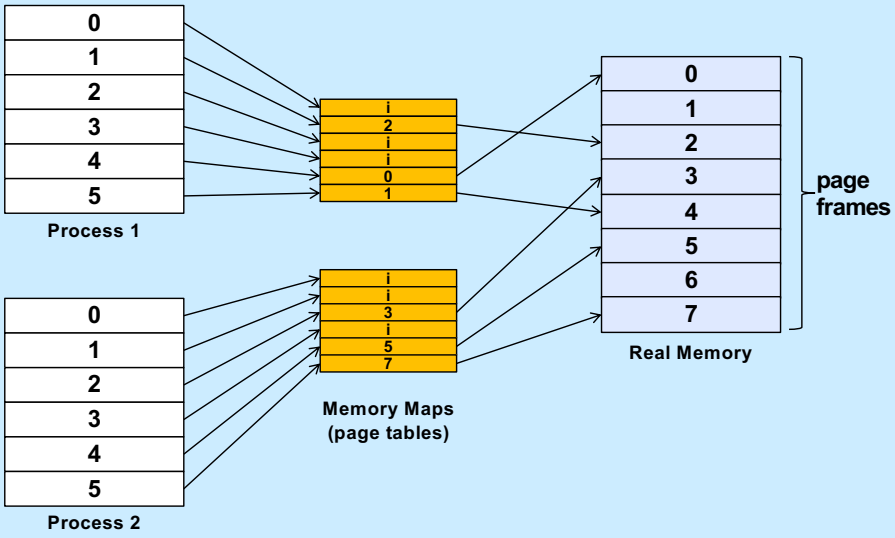
A problem with the one-handed version is that, in systems with large amounts of primary storage, it might take too long for the page-out thread to work its way all around the list of page frames before it can recognize that a page has not been recently referenced. In the two-handed version of the clock algorithm, the page-out thread implements a second hand some distance behind the first. The front hand simply clears reference bits. The second (back) hand removes those pages whose reference bits have not been set to one by the time the hand reaches the page frame.

Why is virtual memory used?

More VM than RM

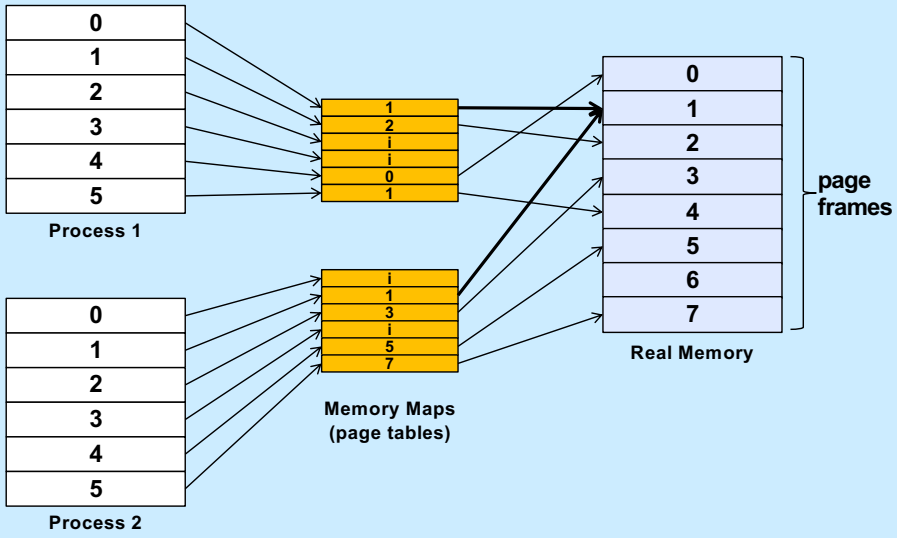


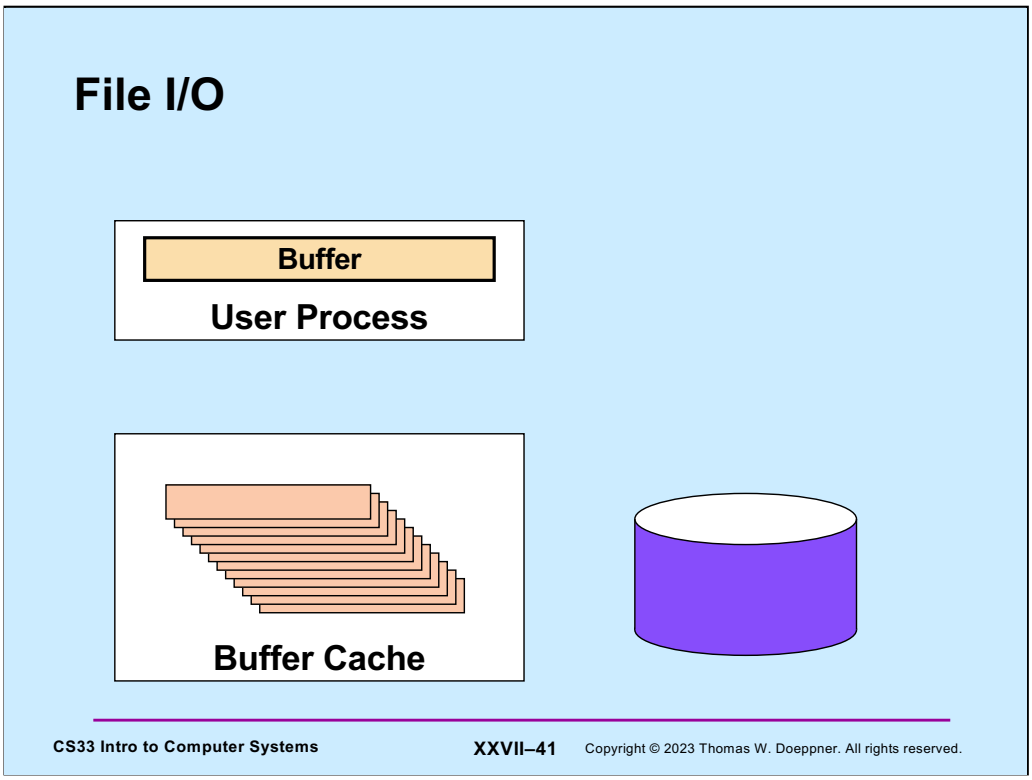
Isolation



Virtual Memory

Sharing

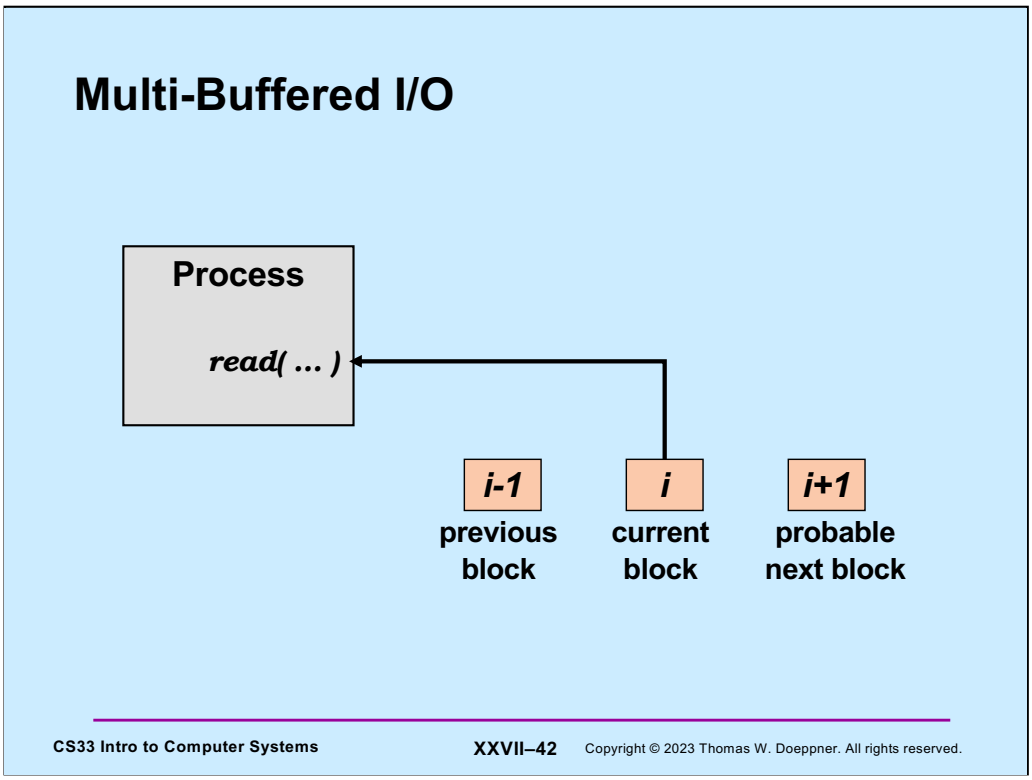




File I/O in Unix, and in most operating systems, is not done directly to the disk drive, but through intermediary buffers, known as the buffer cache, in the operating system's address space. This cache has two primary functions. The first, and most important, is to make possible concurrent I/O and computation within a Unix process. The second is to insulate the user from physical disk-block boundaries.

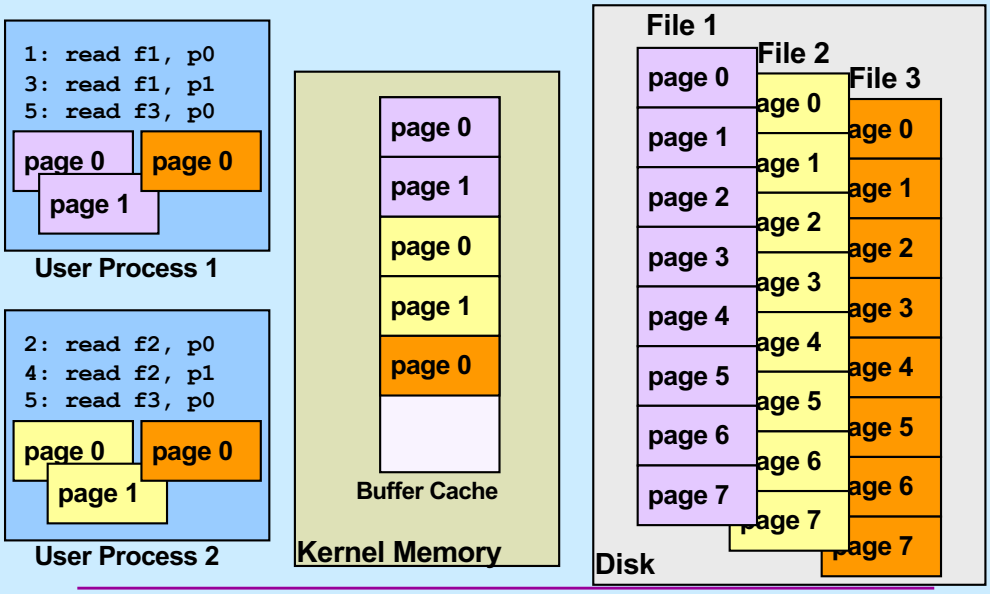
From a user process's point of view, I/O is **synchronous**. By this we mean that when the I/O system call returns, the system no longer needs the user-supplied buffer. For example, after a write system call, the data in the user buffer has either been transmitted to the device or copied to a kernel buffer — the user can now scribble over the buffer without affecting the data transfer. Because of this synchronization, from a user process's point of view, no more than one I/O operation can be in progress at a time.

The buffer cache provides a kernel implementation of multibuffered I/O, and thus concurrent I/O and computation are made possible.

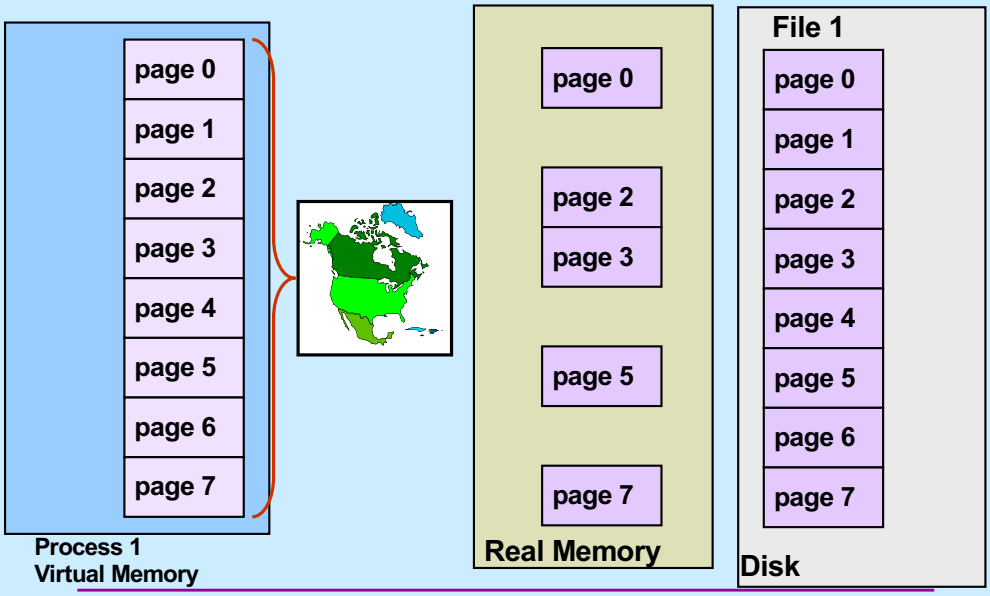


The use of **read-aheads** and **write-behinds** makes possible concurrent I/O and computation: if the block currently being fetched is block *i* and the previous block fetched was block *i-1*, then block *i+1* is also fetched. Modified blocks are normally written out not synchronously but instead sometime after they were modified, asynchronously.

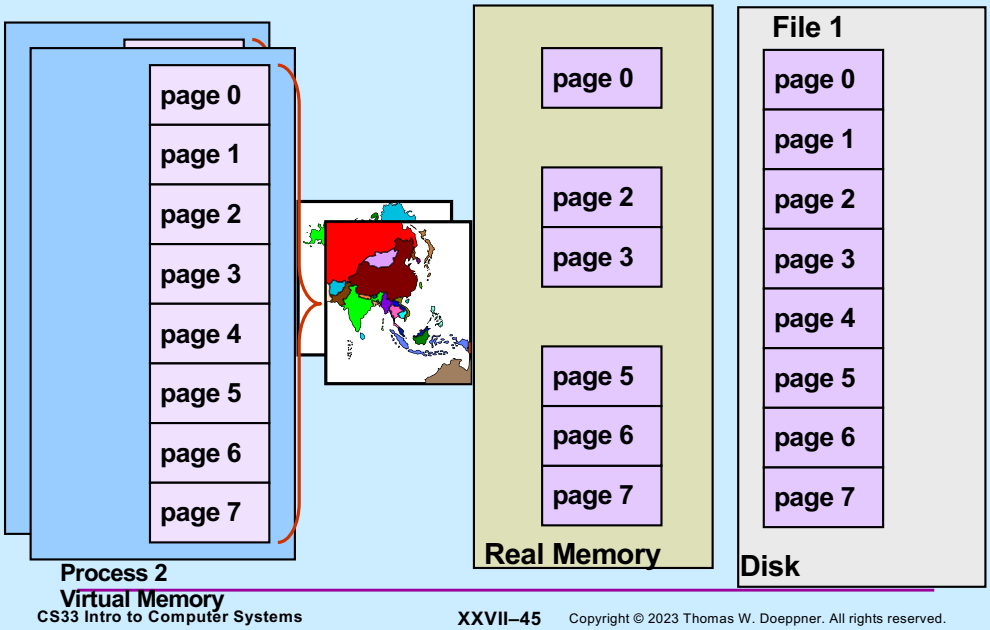
Traditional I/O



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**

```
record_t *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.

Mmap System Call

```
void *mmap(
  void *addr,
    // where to map file (0 if don't care)
  size_t len,
    // how much to map
  int prot,
    // memory protection (read, write, exec.)
  int flags,
    // shared vs. private, plus more
  int fd,
    // which file
  off_t off
    // starting from where
  );
```

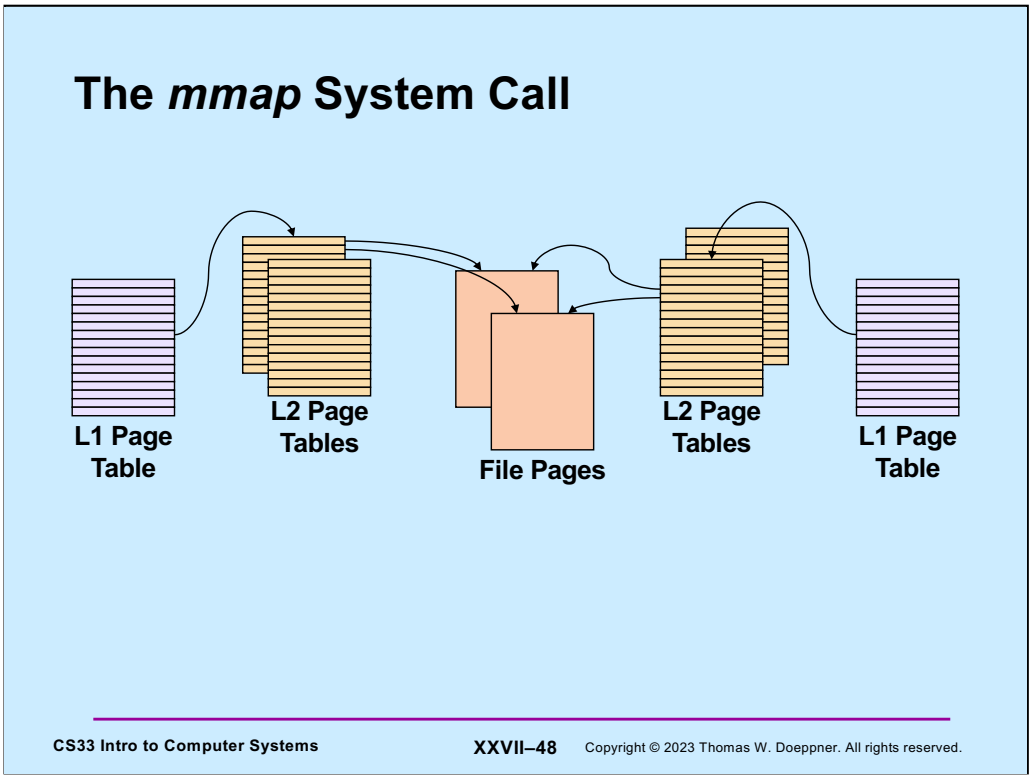

Mmap maps the file given by **fd**, starting at position **off**, for **len** bytes, into the caller's address space starting at location **addr**

- **len** is rounded up to a multiple of the page size
- **off** must be page-aligned
- if **addr** is zero, the kernel assigns an address
- if **addr** is positive, it is a suggestion to the kernel as to where the mapped file should be located (it usually will be aligned to a page). However, if **flags** includes MAP_FIXED, then **addr** is not modified by the kernel (and if its value is not reasonable, the call fails)
- the call returns the address of the beginning of the mapped file

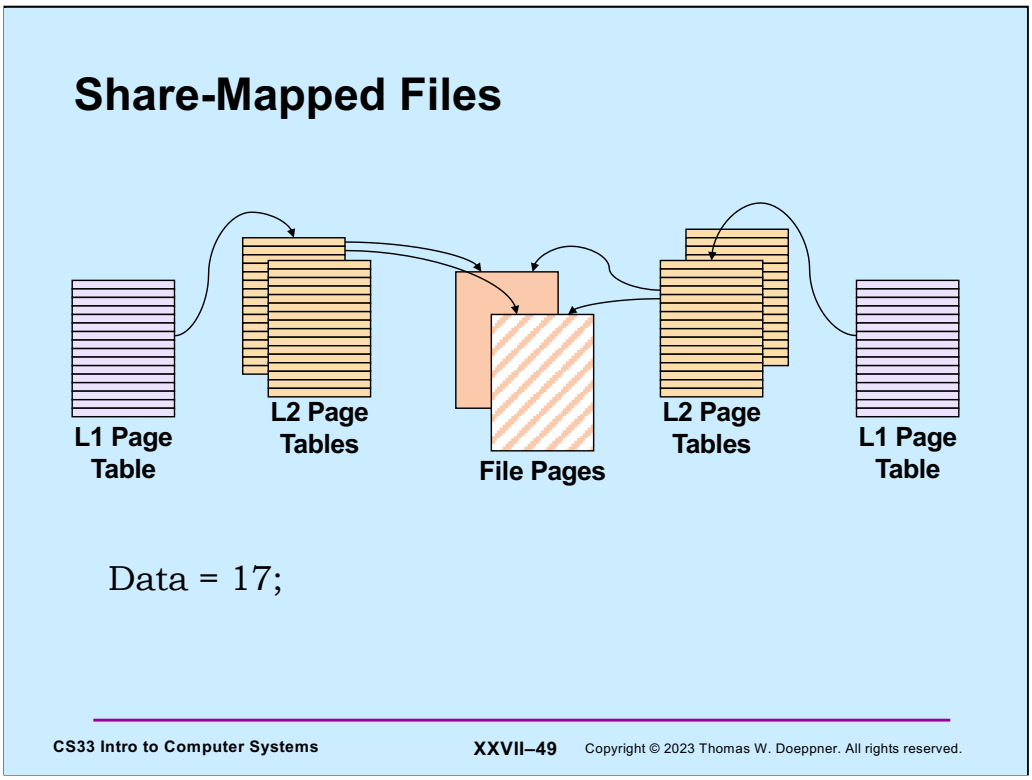
The **flags** argument must include either MAP_SHARED or MAP_PRIVATE (but not both). If it's MAP_SHARED, then the mapped portion of the caller's address space contains the current contents of the file; when the mapped portion of the address space is modified by the process, the corresponding portion of the file is modified.

However, if **flags** includes MAP_PRIVATE, then the idea is that the mapped portion of the address space is initialized with the contents of the file, but that changes made to the mapped portion of the address space by the process are private and not written back to the file. The details are a bit complicated: as long as the mapping process does not modify any of the mapped portion of the address space, the pages contained in it contain the current contents of the corresponding pages of the file. However, if the process modifies a page, then that particular page no longer contains the current contents of the corresponding file page, but contains whatever modifications are made to it by the process. These changes are not written back to the file and not shared with any other process that has mapped the file. It's unspecified what the situation is for other pages in the mapped region after one of them is modified. Depending on the implementation, they might continue to contain the current contents of the corresponding pages of the file until they, themselves, are modified. Or they might also be treated as if they'd just been written to and thus

no longer be shared with others.



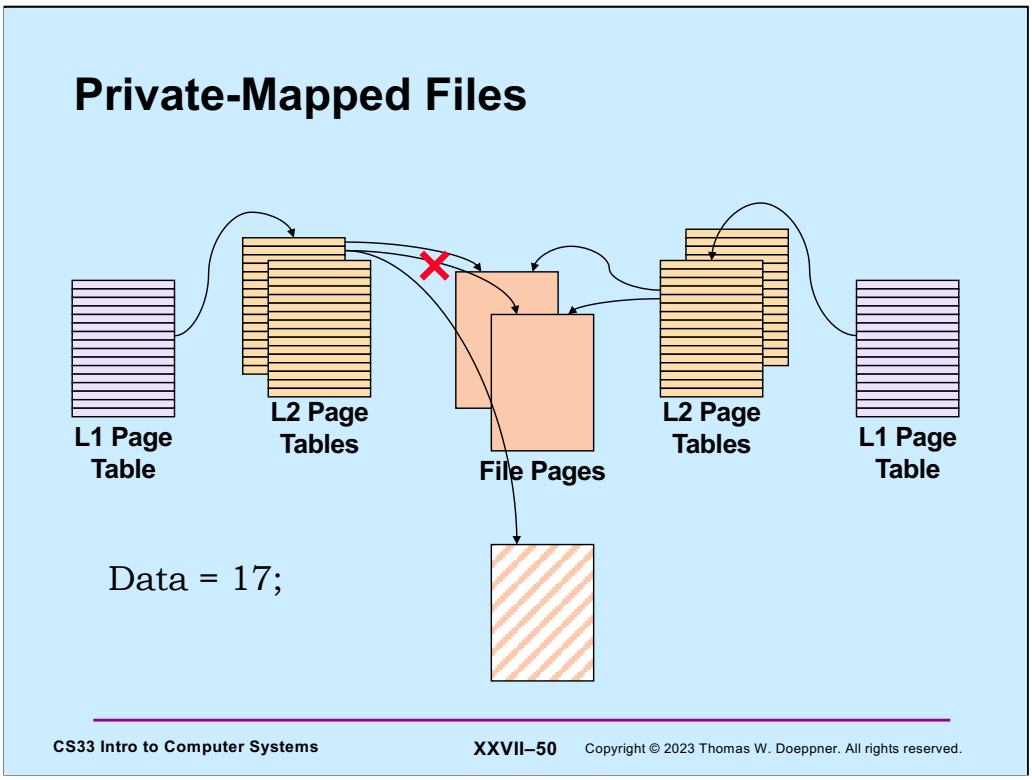
The **mmap** system call maps a file into a process's address space. All processes mapping the same file can share the pages of the file.



Here, **Data** is a variable located in the highlighted file page.

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.

Hence, the change to **Data** is seen by both processes mapping 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 **Data** (in the private-mapped portion of the address space) are made to a copy of the page containing Data. Thus, the other process will continue to see the original Data.

Example

```
int main( ) {
  int fd;
  dataObject_t *dataObjectp;

  fd = open("file", O_RDWR);
  if ((int)(dataObjectp = (dataObject_t *)mmap(0,
      sizeof(dataObject_t),
      PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0)) == -1) {
    perror("mmap");
    exit(1);
  }

  // dataObjectp points to region of (virtual) memory
  // containing the contents of the file

  ...

}
```


Here we map the contents of a file containing a dataObject_t into the caller's address space, allowing it both read and write access. Note mapping the file into memory does not cause any immediate I/O to take place. The operating system will perform the I/O when necessary, according to its own rules.

fork and mmap

```
int main() {
  int x=1;

  if (fork() == 0) {
    // in child
    x = 2;
    exit(0);
  }
  // in parent
  while (x==1) {
    // will loop forever
  }
  return 0;
}
```

```
int main() {
  int fd = open( ... );
  int *xp = (int *)mmap(...,
      MAP_SHARED, fd, ...);
  xp[0] = 1;
  if (fork() == 0) {
    // in child
    xp[0] = 2;
    exit(0);
  }
  // in parent
  while (xp[0]==1) {
    // will terminate
  }
  return 0;
}
```


When a process calls fork and creates a child, the child's address space is normally a copy of the parent's. Thus changes made by the child to its address space will not be seen in the parent's address space (as shown in the left-hand column). However, if there is a region in the parent's address space that has been mmapped using the MAP_SHARED flag, and subsequently the parent calls fork and creates a child, the mmapped region is not copied but is shared by parent and child. Thus changes to the region made by the child will be seen by the parent (and vice versa).