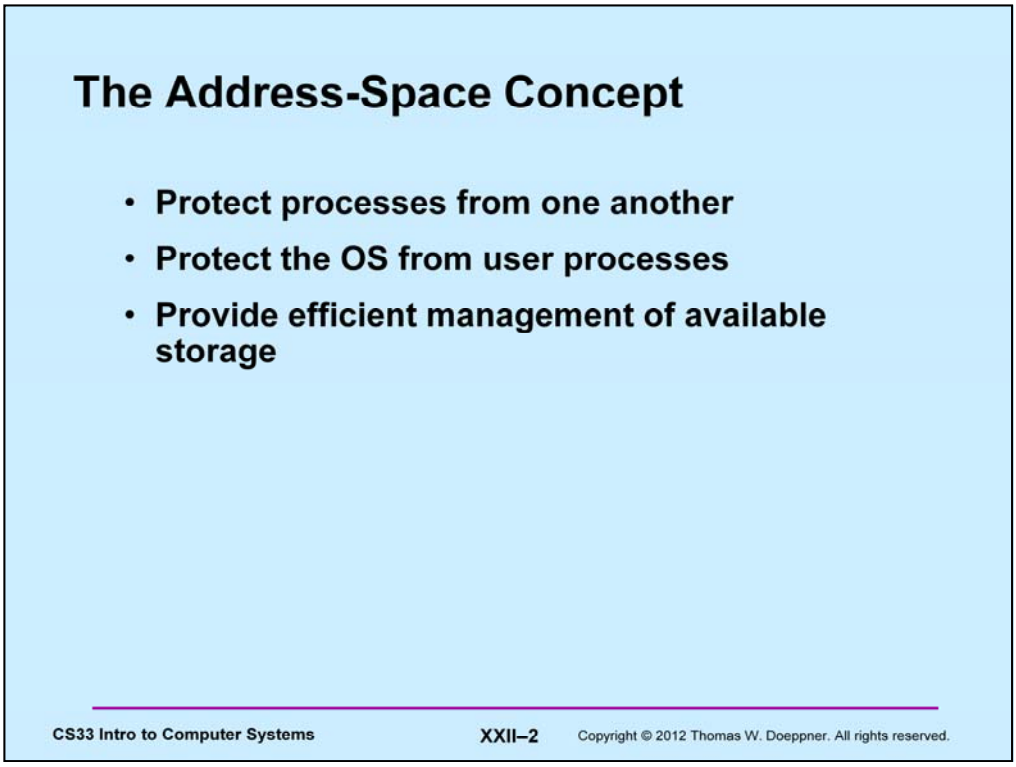
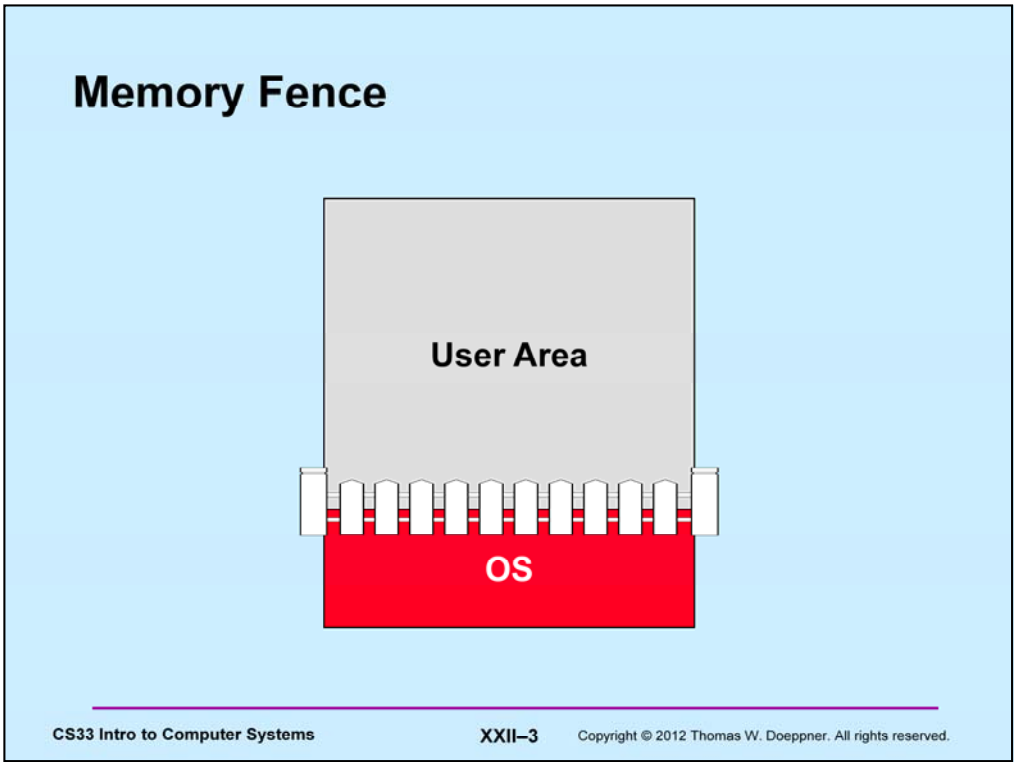
CS 33

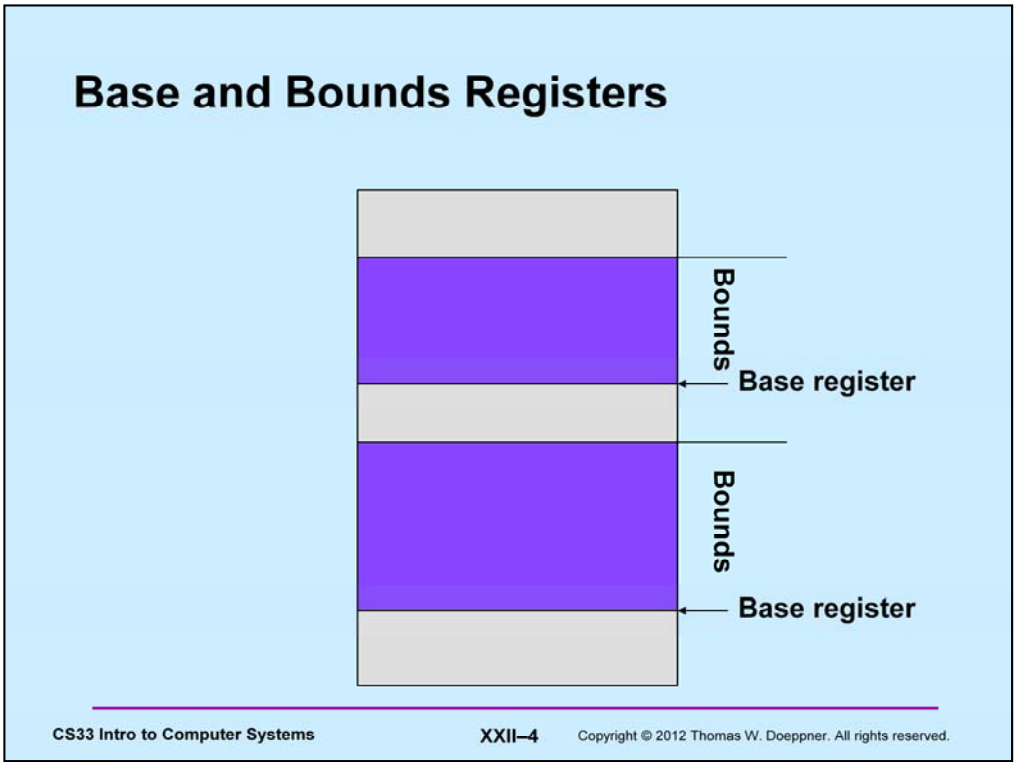
Virtual Memory (1)



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.

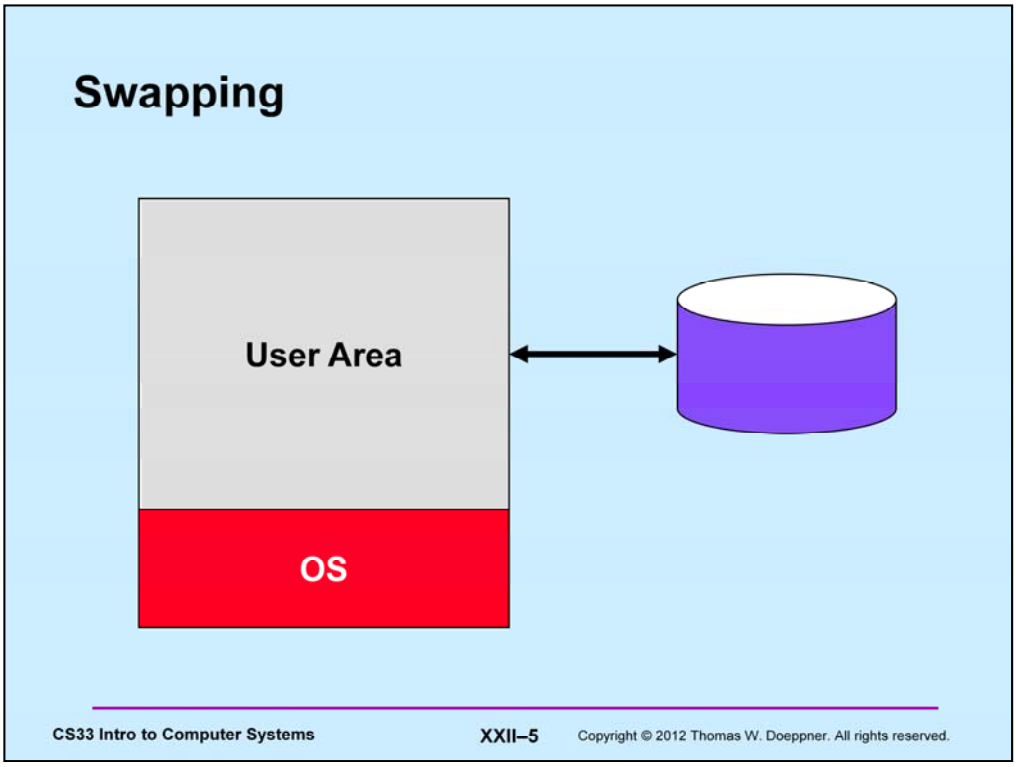


Early approaches to managing the address space were concerned primarily with protecting the operating system from the user. One technique was the hardware-supported concept of the *memory fence*: an address was established below which no user mode access was allowed. The operating system was placed below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



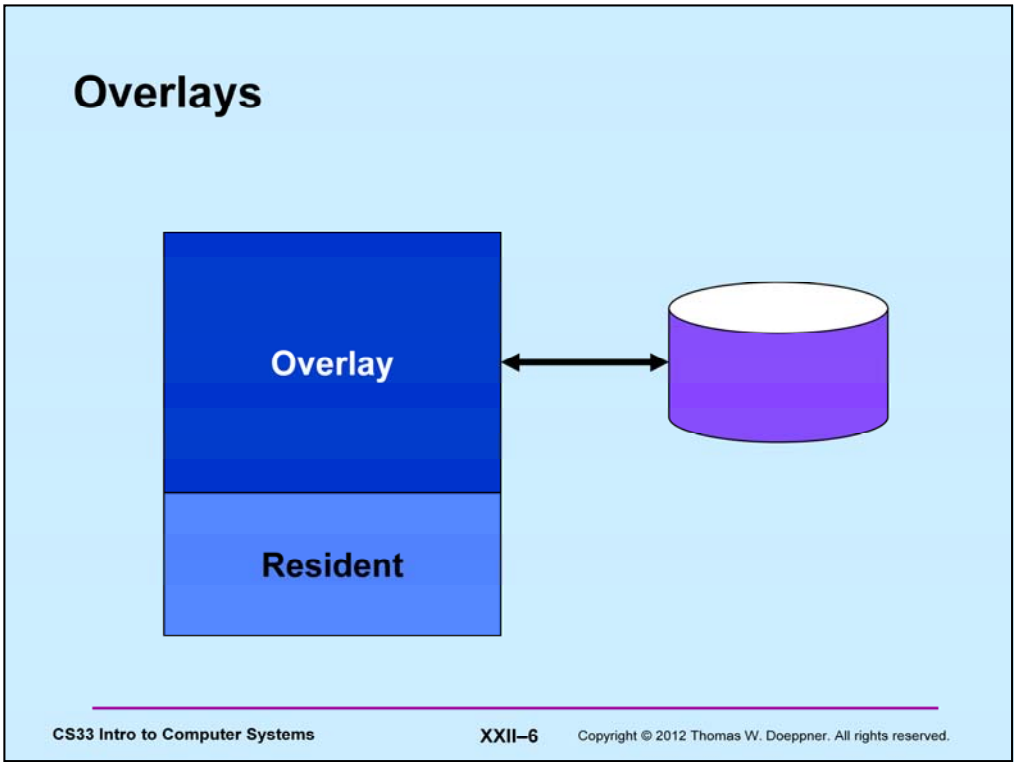
The memory-fence approach protected the operating system, but did not protect user processes from one another. (This wasn't an issue for many systems—there was only one user process at a time.) Another technique, still employed in some of today's systems, is the use of *base and bounds registers* to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



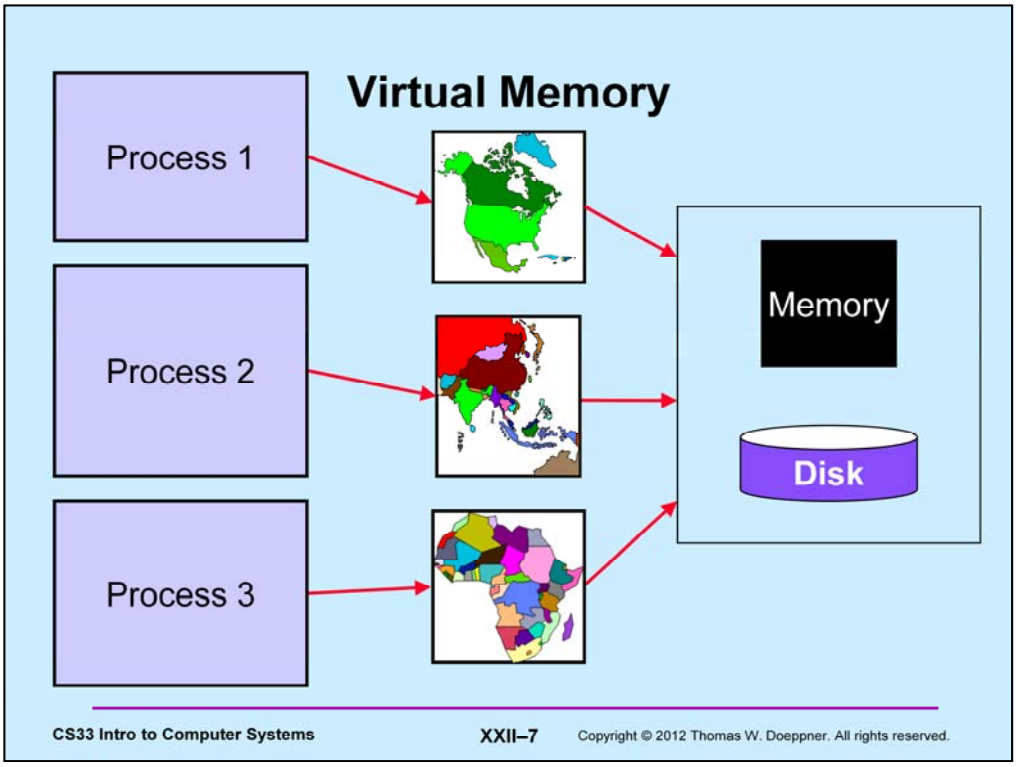
Swapping is a technique, still in use today, in which the images of entire processes are transferred back and forth between primary and secondary storage. An early use of it was for (slow) time-sharing systems: when a user paused to think, his or her process was swapped out and that of another user was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

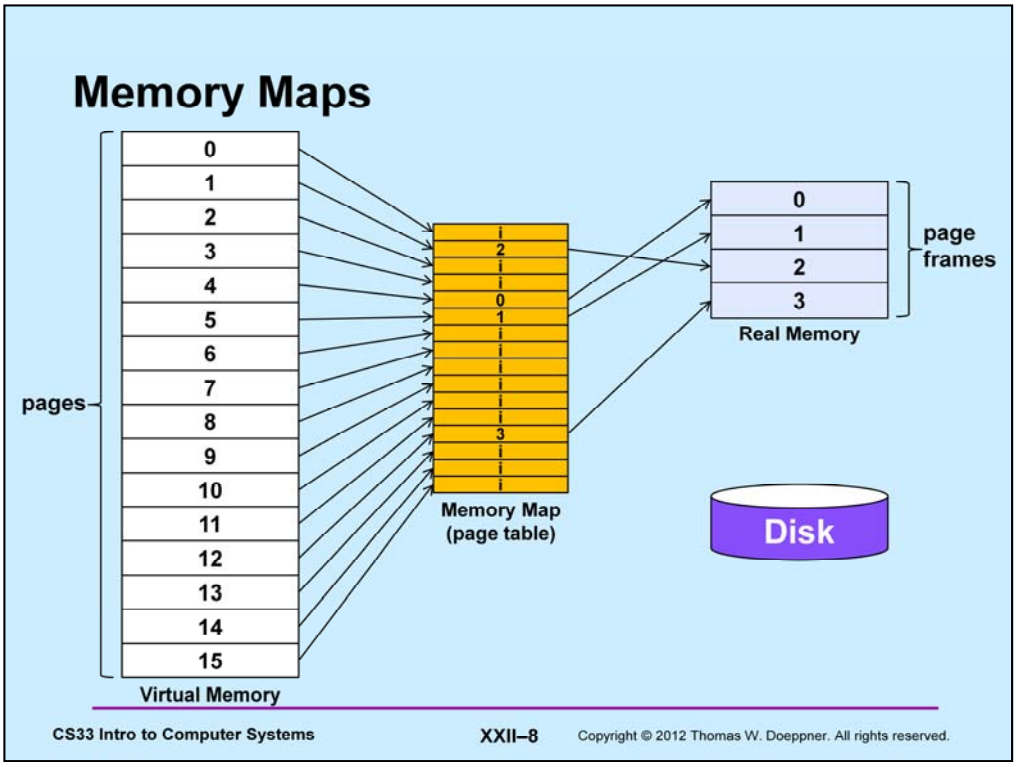
The advantage of this technique is that the programmer has complete control of the use of memory and can make the necessary optimization decisions. The disadvantage is that the programmer *must* make the necessary decisions to make full use of memory (the operating system doesn't help out). Few programmers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



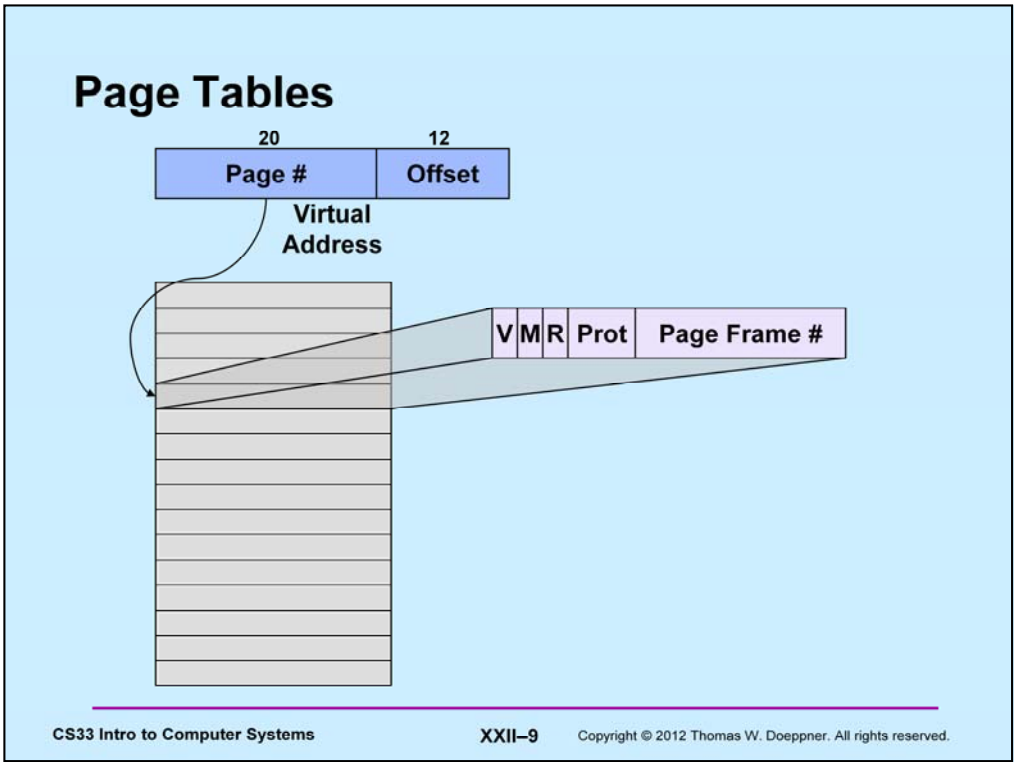
One way to look at virtual memory is as an automatic overlay technique: processes "see" an address space that is larger than the amount of real memory available to them; the operating system is responsible for the overlaying.

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 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.



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.



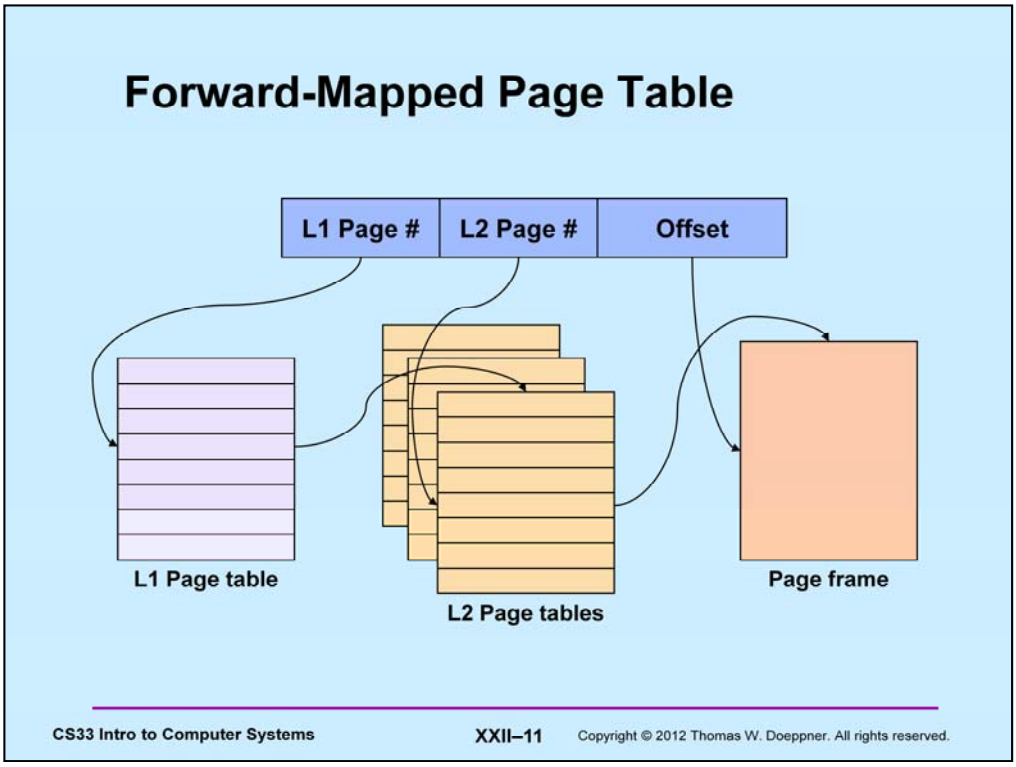
A page table is an array of *page table entries*. Suppose we have, as is the usual case for the x86, a 32-bit virtual address and a page size of 4096 bytes. The 32-bit address might be split into two parts: a 20-bit *page number* and a 12-bit *offset* within the page. When a thread generates an address, 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.

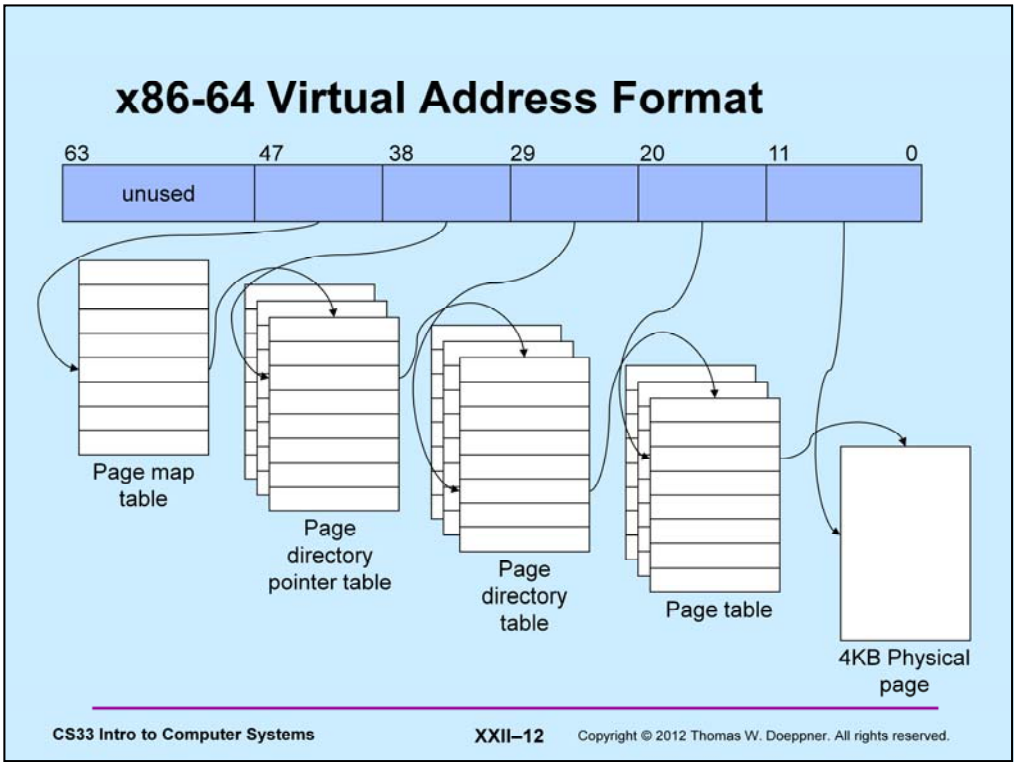
Page-Table Size

- 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 $6 worth of memory
- 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 worth of memory ...

In the not-all-that-distant past, 4 megabytes of memory would have cost several million dollars.



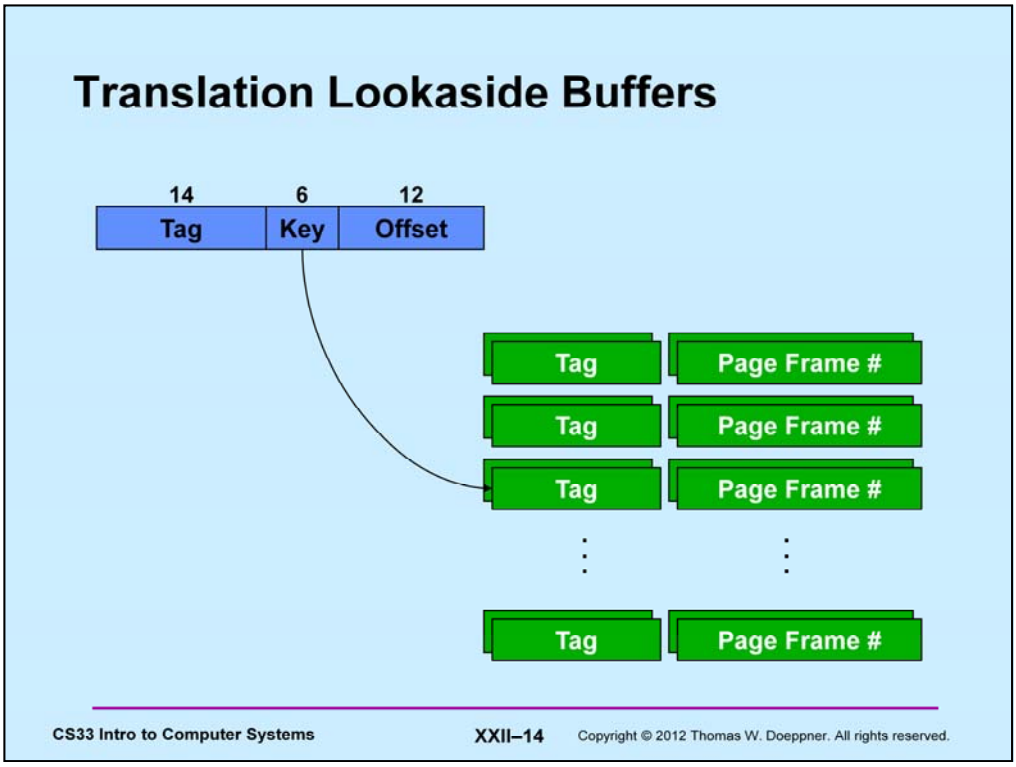
The forward-mapped, or two-level page table provides a means for reducing the memory requirements of the address map. All of the L1 page must reside in real memory when the address space is in use, but it is relatively small. Though there are a large number of L2 page tables (each relatively small), only a few need to reside in real memory at once. For the x86, both the L1 and L2 page numbers are 10 bits long, while the offset is 12 bits long.



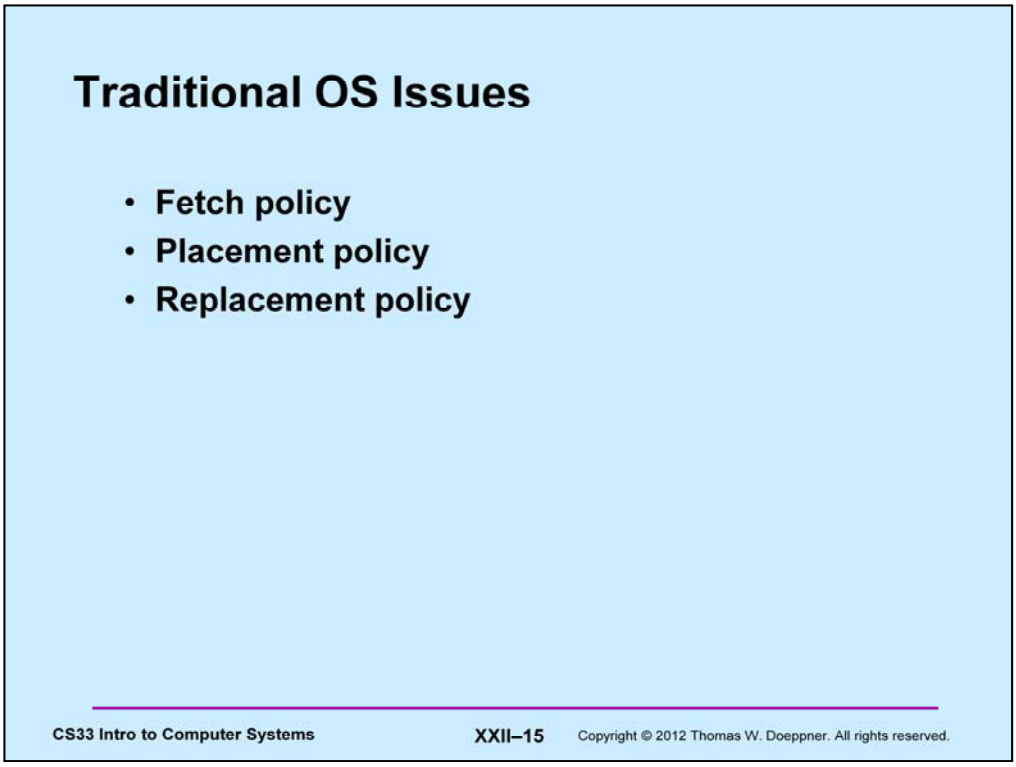
For the x86-64, four levels of translation are done (the high-order 16 bits of the address are not currently used), thus it really supports "only" a 48-bit address space. Note that only the "page map table" must reside in real memory at all times. The other tables must be resident only when necessary.

Performance

- Page table resides in real memory (DRAM)
- A 32-bit virtual-to-real translation requires two accesses 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!



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, as discussed in lecture 16.



The operating-system issues related to paging are traditionally split into three areas: the *fetch policy*, which governs when pages are fetched from secondary storage and which pages are fetched, the *placement policy*, which governs where the fetched pages are placed in primary storage, and the *replacement policy*, which governs when and which pages are removed from primary storage (and perhaps written back to secondary storage).

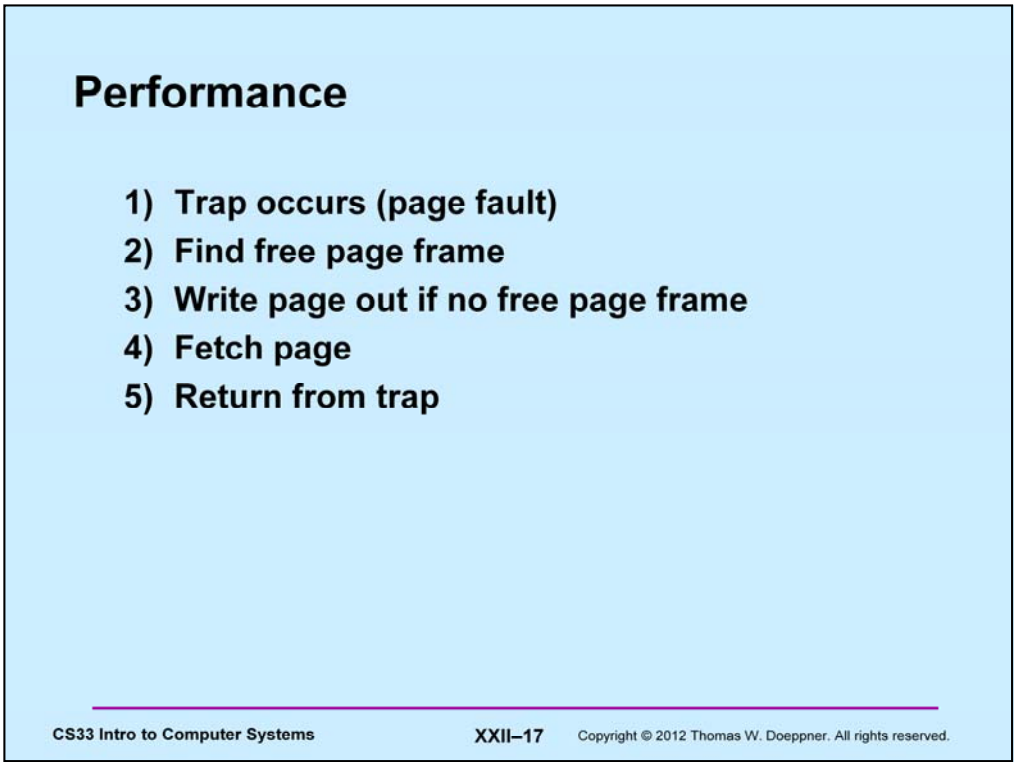
A Simple Paging Scheme

- Fetch policy
  - start process off with no pages in primary storage
  - bring in pages on demand (and only on demand) (this is known as demand paging)
- Placement policy
  - it doesn't matter—put the incoming page in the first available page frame
- Replacement policy
  - replace the page that has been in primary storage the longest (FIFO policy)

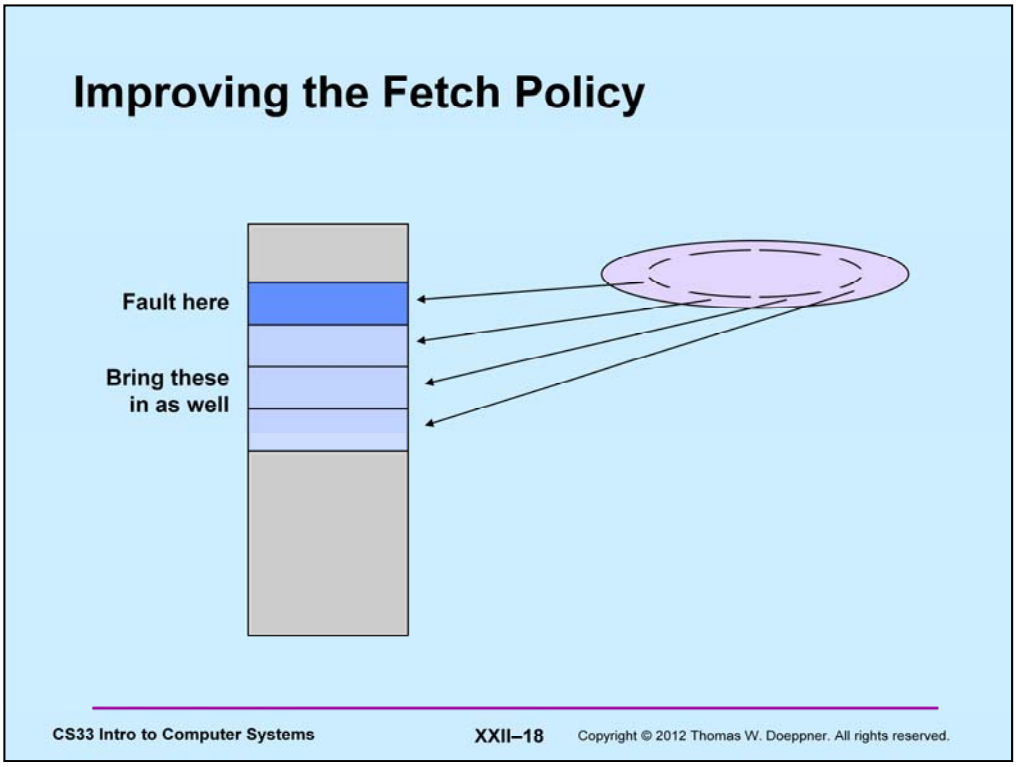
Let's first consider a fairly simple scheme for paging. The fetch policy is based on demand paging: pages are fetched only when needed. So, when we start a process, none of its pages are in primary memory and pages are fetched in response to page faults. The idea is that, after a somewhat slow start, the process soon has enough pages in memory that it can execute fairly quickly. The big advantage of this approach is that no primary storage is wasted on unnecessary pages.

As is usually the case, the placement policy is irrelevant—it doesn't matter where pages are placed in primary storage.

We start with a much too simple-minded replacement policy: when we want to fetch a page, but there is no room for it, we replace the page that has been in primary memory the longest. (This is known as a FIFO or first-in-first-out approach.)



To get an idea how well our paging scheme will perform, let's consider what happens in response to a page fault. The steps are outlined in the picture. How expensive are these steps? Clearly step 1 (combined with step 2) incurs a fair amount of expense—a reference to memory might take, say, 500 nanoseconds if there is no page fault. If there is a page fault, even if no I/O is performed, the time required before the faulted instruction can be resumed is at least tens of microseconds. If a page frame is available, then step 2 is very quick, but if not, then the faulting thread must free a page, which may result in an output operation that could take many milliseconds. Then, in step 4, the thread must wait for an input operation to complete, which could take many more milliseconds.



One way to improve the performance of our paging system is to reduce the number of page faults. Ideally, we'd like to be able to bring a page into primary storage before it is referenced (but only bring in those pages that will be referenced soon). In certain applications this might be possible, but for the general application, we don't have enough knowledge of what the program will be doing. However, we can make the reasonable assumption that programs don't reference their pages randomly, and furthermore, if a program has just referenced page *i*, there is a pretty good chance it might soon reference page *i+1*. We aren't confident enough of this to go out of our way to fetch page *i+1*, but if it doesn't cost us very much to fetch this next page, we might as well do it.

We know that most of the time required for a disk transfer is spent in seek and rotational latency delays—the actual data transfer takes relatively little time. So, if we are fetching page *i* from disk and page *i+1* happens to be stored on disk adjacent to page *i*, then it does not require appreciably more time to fetch both page *i* and page *i+1* than it does to fetch just page *i*. Furthermore, if page *i+2* follows page *i+1*, we might as well fetch it too.

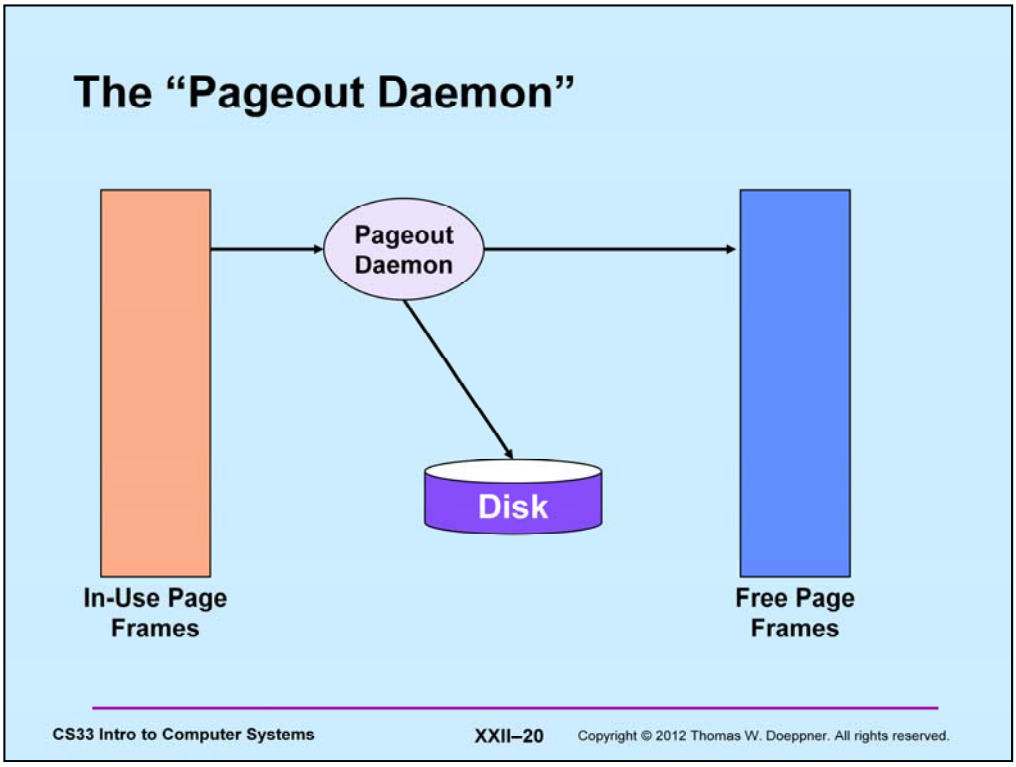
By using this approach, we are effectively increasing the page size—we are grouping hardware pages into larger, operating-system pages. However, we only fetch more pages than are required if there are free page frames available. Furthermore, we can arrange so that such "pre-paged" pages are the first candidates to be replaced if there is a memory shortage and these pages have not been referenced.

Improving the Replacement Policy

- When is replacement done?
  - doing it "on demand" causes excessive delays
  - should be performed as a separate, concurrent activity
- Which pages are replaced?
  - FIFO policy is not good
  - want to replace those pages least likely to be referenced soon

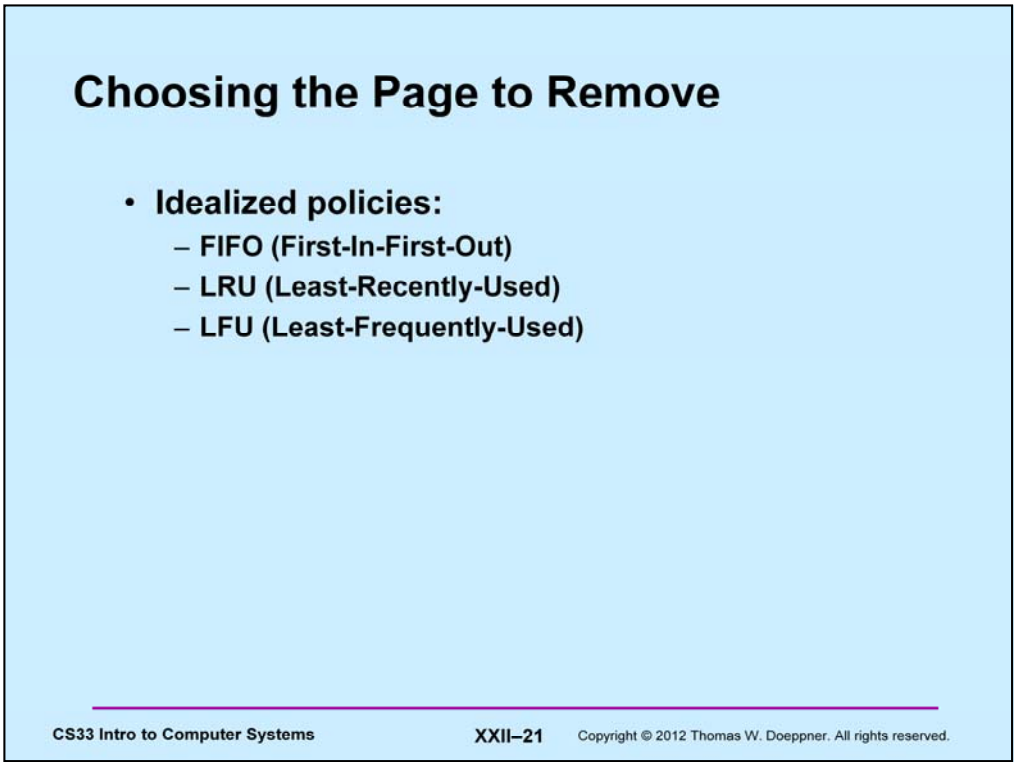
The replacement policy of our simple paging scheme left much to be desired. The first problem with it is that we wait until we are totally out of free page frames before starting to replace pages, and then we only replace one page at a time. A better technique might be to have a separate thread maintain the list of free page frames. Thus, as long as this thread provides an adequate supply of free pages, no faulting thread ever has to wait for a page to be written out.

The other problem with the replacement policy is the use of the FIFO technique for deciding which pages to remove from primary storage—the page that has been in memory the longest could well be the page that is getting the vast majority of the references. Ideally, we would like to remove from primary storage that page whose next reference will be the farthest in the future.



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 if there is a 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.

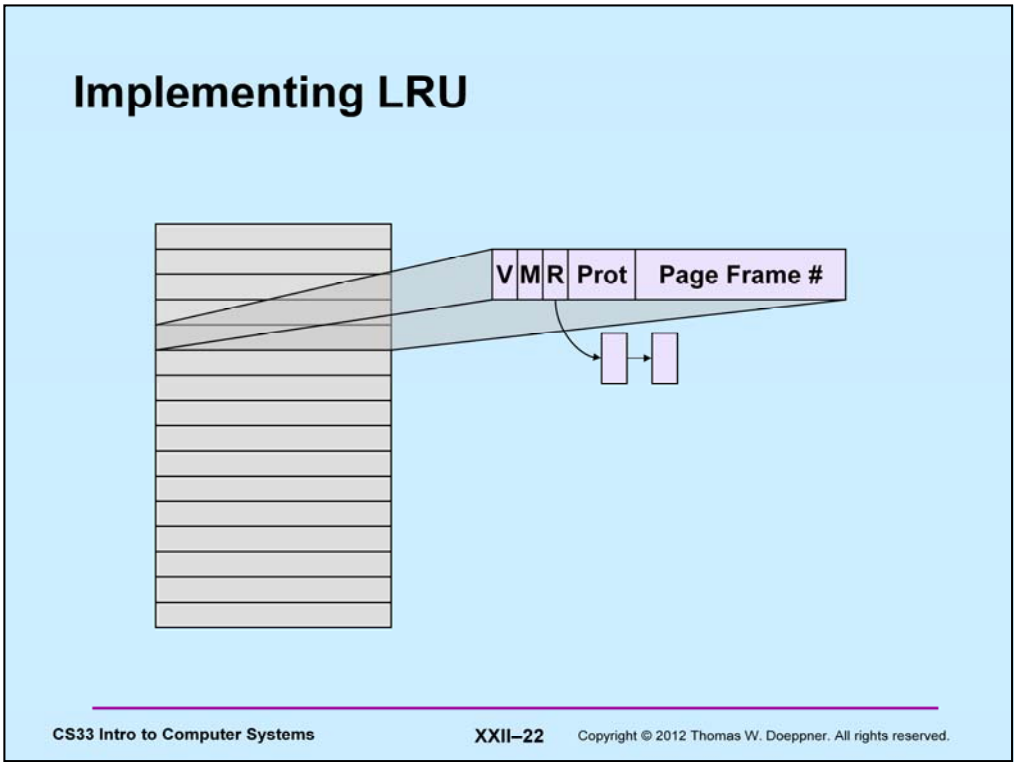


Determining which pages to replace is a decision that could have significant ramifications to system performance. If the wrong pages are replaced, i.e. pages which are referenced again fairly soon, then not only must these pages be fetched again, but a new set of pages must be tagged for removal.

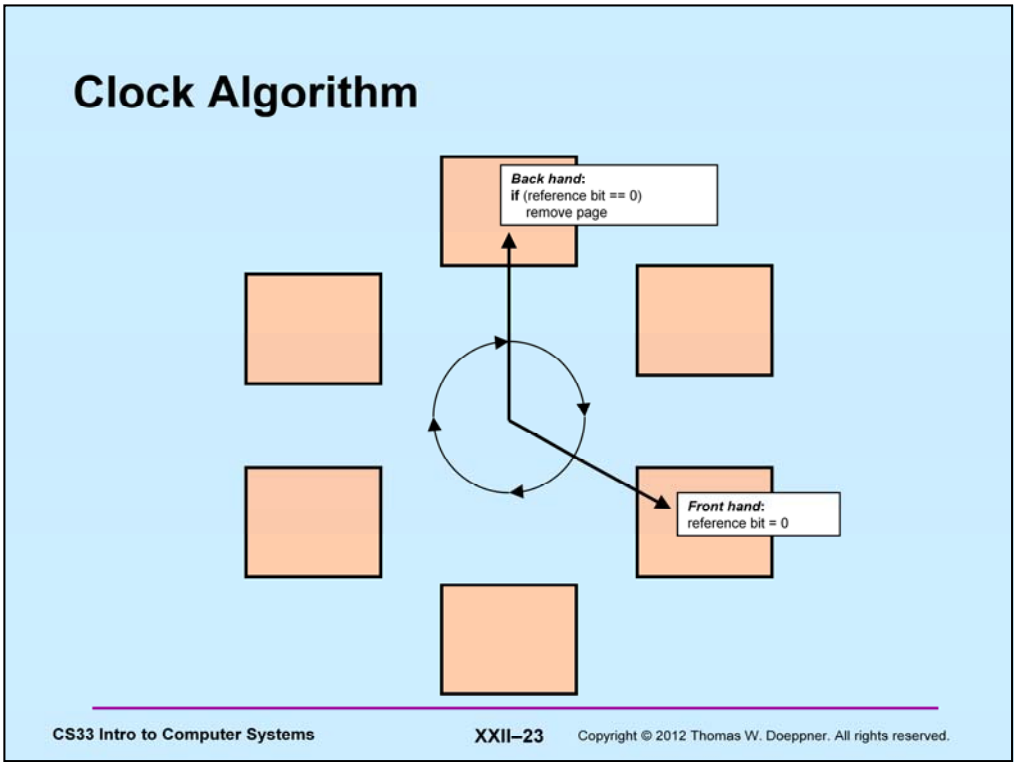
The FIFO scheme has the benefit of being easy to implement, but often removes the wrong pages. Now, we can't assume that we have perfect knowledge of a process's reference pattern to memory, but we can assume that most processes are reasonably "well behaved." In particular, it is usually the case that pages referenced recently will be referenced again soon, and that pages that haven't been referenced for a while won't be referenced for a while. This is the basis for a scheme known as LRU—least recently used. We order pages in memory by the time of their last access; the next page to be removed is the page whose last access was the farthest in the past.

A variant of this approach is LFU—least frequently used. Pages are ordered by their frequency of use, and we replace the page which has been used the least frequently of those still in primary storage.

Both these latter two policies have the advantage that they behave better for average programs than FIFO, but the disadvantage that they are costly to implement, since they would involve keeping track of every reference to memory.

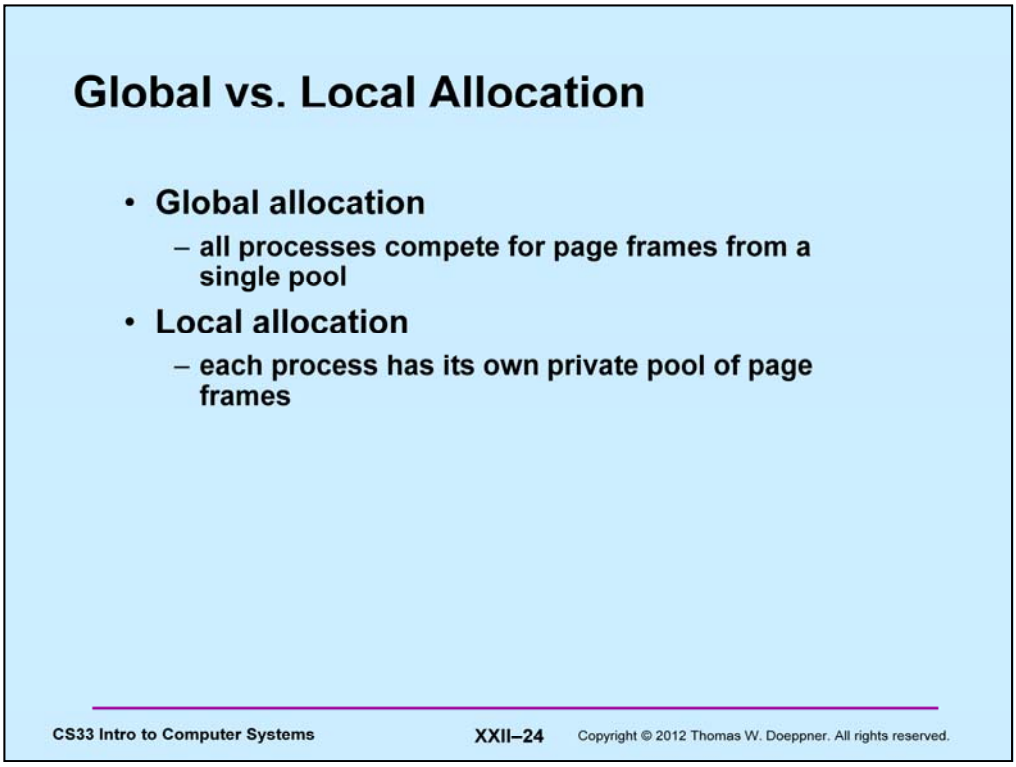


The standard approach for implementing the LRU (and LFU) strategy is to approximate it. We divide time into "epochs" and order pages by whether or not they were referenced in successive epochs. To do this, we rely upon a hardware-supported *reference bit*, which is set whenever a page is referenced. Periodically we can copy these bits to per-page-frame data structures, then clear the bits in the page table entries. Depending upon how much history we wish to maintain, each per-page-frame data structure might keep the recent reference bits in a shift vector. If the shifts are to the right, then the most recently referenced pages have ones in the left bits of their shift vectors. (In practice, such shift vectors contain one or two bits.)



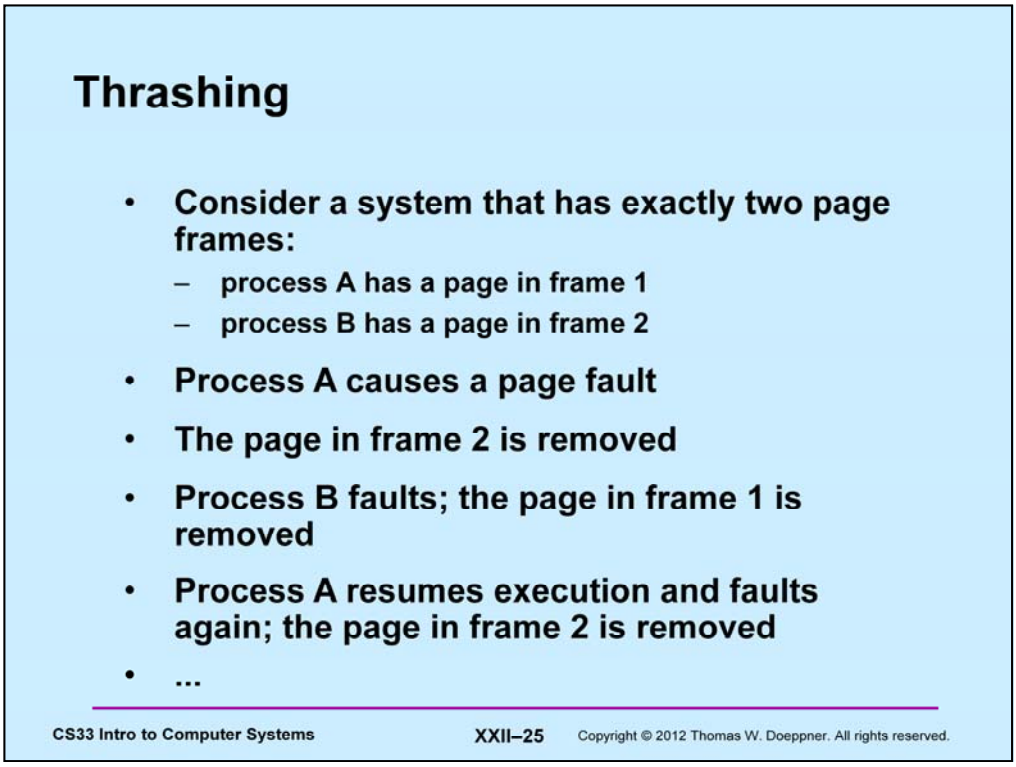
Rather than implement the approximate LRU algorithm exactly as we've described it, many systems use a continual approach known as the clock algorithm. All active page frames are conceptually arranged in a circularly linked list. The page-out thread slowly traverses the list. The "one-handed" version 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.



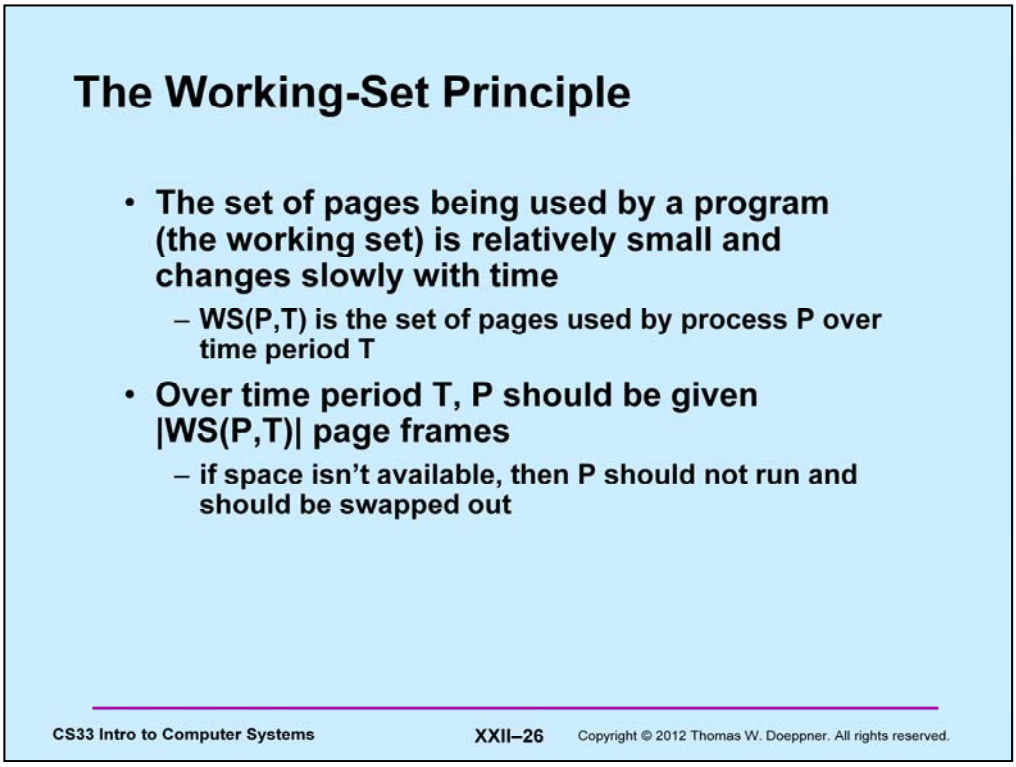
Another paging-related problem is the allocation of page frames among different processes (i.e. address spaces). There are two simple approaches to this. The first is to have all processes compete with one another for page frames. Thus, processes that reference a lot of pages tend to have a lot of page frames, and processes that reference fewer pages have fewer page frames.

The other approach is to assign each process a fixed pool of page frames, thereby avoiding competition. This has the benefit that the actions of one process don't have quite such an effect on others, but leaves the problem of determining how many page frames to give to each process.



The main argument against the global allocation of page frames is thrashing. The situation described in the picture is an extreme case, but it illustrates the general problem. If demand for page frames is too great, then while one process is waiting for a page to be fetched, the pages it already has are "stolen" from it to satisfy the demands of other processes. Thus when this first process finally resumes execution, it immediately faults again.

A symptom of a thrashing situation is that the processor is spending a significant amount of time doing nothing—all of the processes are waiting on page-in requests. A perhaps apocryphal but amusing story is that the batch monitor on early paging systems would see that the idle time of the processor had dramatically increased and would respond by allowing more processes to run (thus aggravating an already terrible situation).

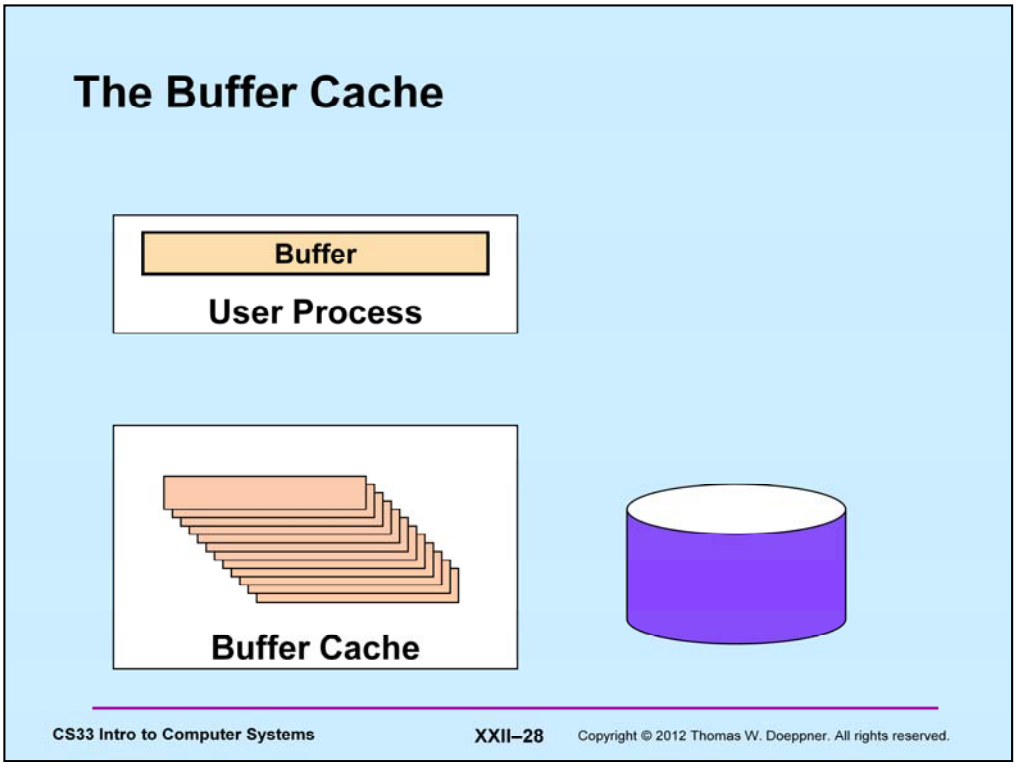


The *working-set principle* is a simple, elegant principle devised by Peter Denning in the late '60s. Assume that, in the typical process, the set of pages being used varies slowly over time (and is a relatively small subset of the complete set of pages in the process's address space). We refer to this set of pages being used by a process as its *working set*. The key to the efficient use of primary storage is to assure that each process is given enough page frames to hold its working set. If we don't have enough page frames to hold each process's working set, then demand for page frame is too heavy and we are in danger of entering a thrashing situation. The correct response in this situation is to remove one or more processes in their entirety (i.e. swap them out), so that there are sufficient page frames for the remaining processes.

Certainly a problem with this principle is that it is imprecise. Over how long a time period should we measure the size of the working set? What is reasonable, suggests Denning, is a time roughly half as long as it takes to fetch a page from the backing store. Furthermore, how do we determine which pages are in the working set? We can approximate this using techniques similar to those used for approximating LRU.

The working-set principle is used in few, if any systems (the size of the working set is just too nebulous a concept). However, it is very important as an ideal to which paging systems are compared.

Virtual Memory and File I/O

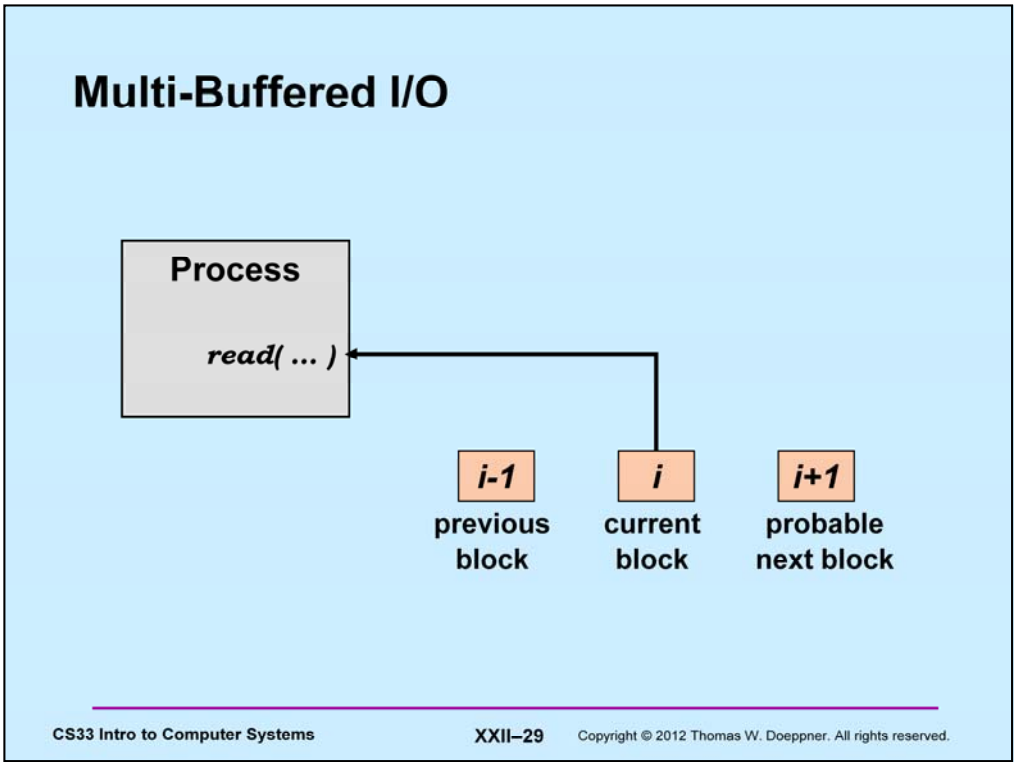


File I/O in Unix, and in most operating systems, is not done directly to the disk drive, but through an intermediary, the *buffer cache*, which is in the operating system's address space.

The buffer 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.

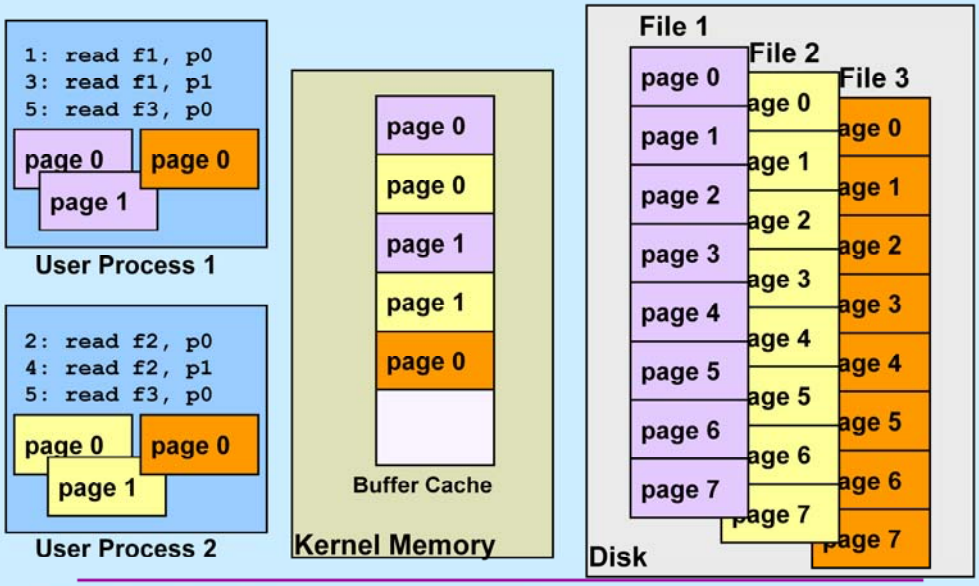
From a user process's point of view, I/O is *synchronous*. By this we mean that when the I/O system call returns, the system no longer needs the user-supplied buffer. For example, after a write system call, the data in the user buffer has either been transmitted to the device or copied to a kernel buffer—the 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.

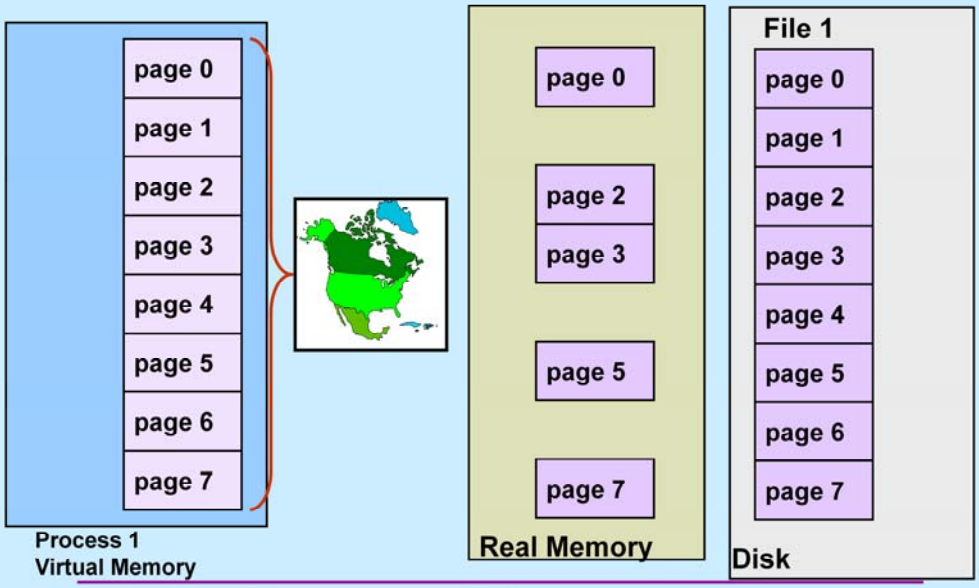


The use of *read-aheads* and *write-behinds* makes possible concurrent I/O and computation: if the block currently being fetched is block *i* and the previous block fetched was block *i-1*, then block *i+1* is also fetched. Modified blocks are normally written out not synchronously but instead sometime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

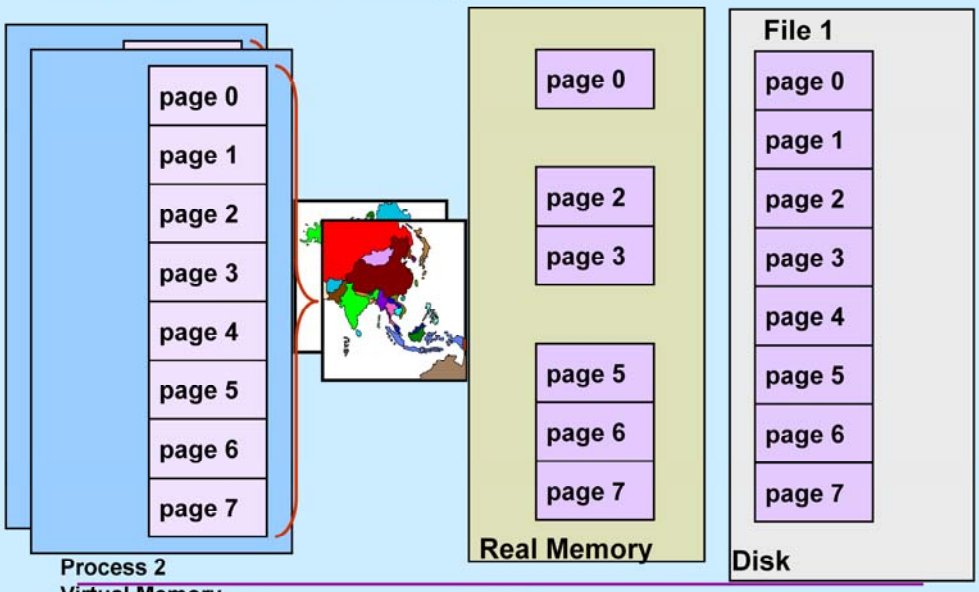
Traditional I/O



Mapped File I/O



Multi-Process Mapped File I/O



Mapped Files

- **Traditional File I/O**

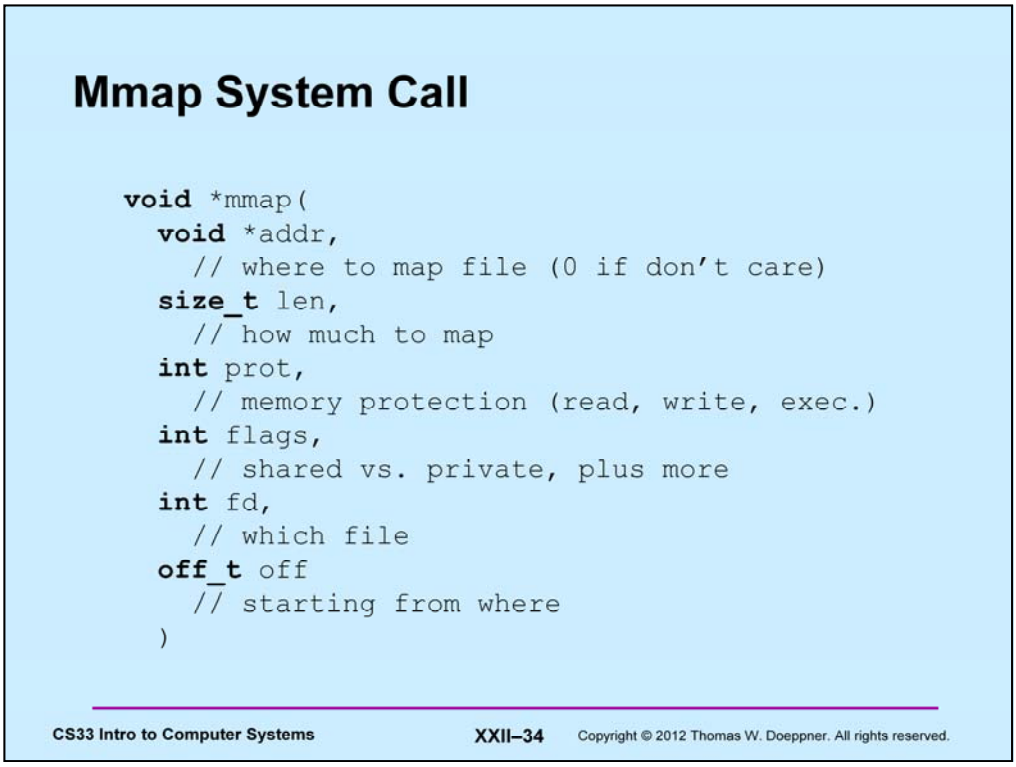
```
char buf[BigEnough];
fd = open(file, O_RDWR);
for (i=0; i<n_recs; i++) {
    read(fd, buf, sizeof(buf));
    use(buf);
}
```

- **Mapped File I/O**

```
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.



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.

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).