Memory Management Part 4

Friday’s Quiz

We’d like to virtualize EPT. Assume that setting EPTP causes a VMexit if done on a VMM that’s not running in real ring -1. What does the VMM running at level 0 (in ring -1) do when it receives such a VMexit from a VMM running at level 1?

a) it sets EPTP to point to the composition of the page tables mapping VMM_0’s address space to real memory and the page tables pointed to by the value being attempted to be put in EPT

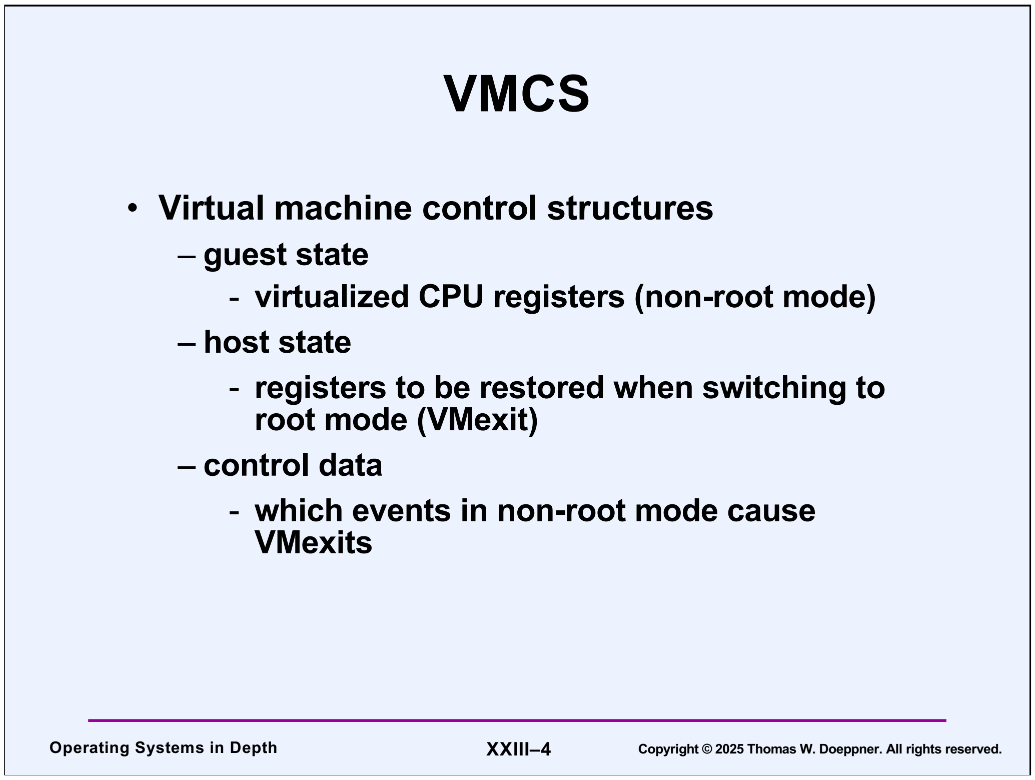
b) nothing: the EPT mechanism is virtualized by the hardware

c) something else

VMX

- New processor mode: root
  - ring -1: root mode
  - rings 0-3: non-root mode
- Certain actions cause processor in non-root mode to switch to root mode
  - VMexit
- When in root mode, processor can switch back to non-root mode
  - VMenter

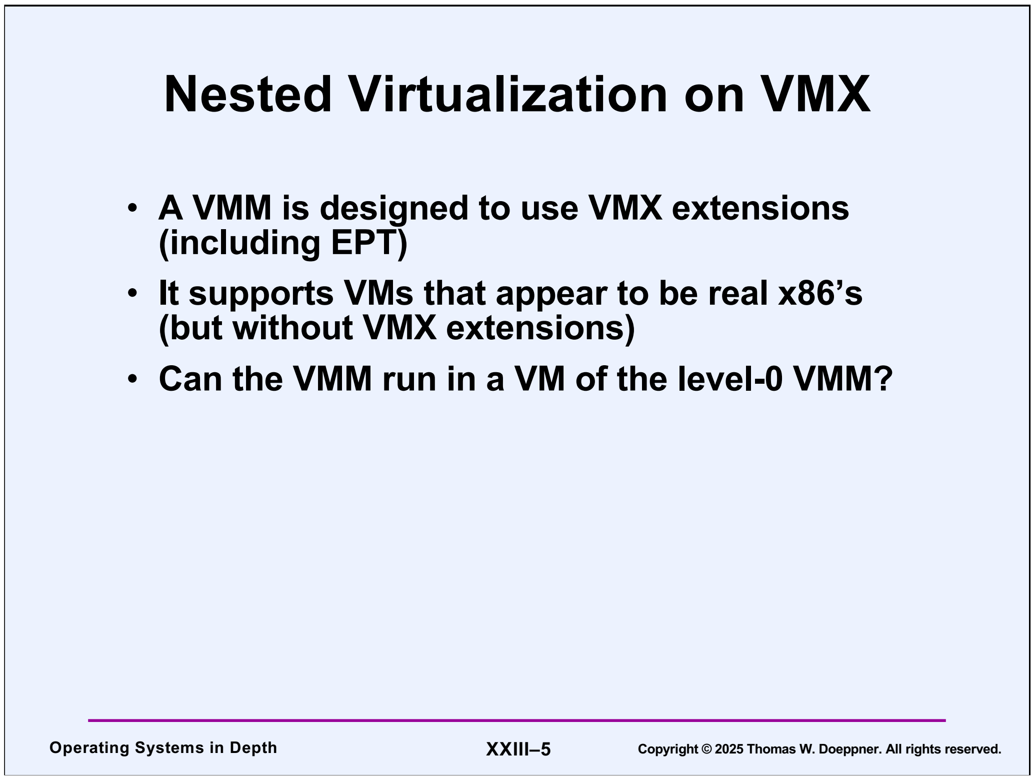
This is a review of what we’ve discussed before.



VMCS

- Virtual machine control structures
  - guest state
    - virtualized CPU registers (non-root mode)
  - host state
    - registers to be restored when switching to root mode (VMexit)
  - control data
    - which events in non-root mode cause VMexits

So is this.

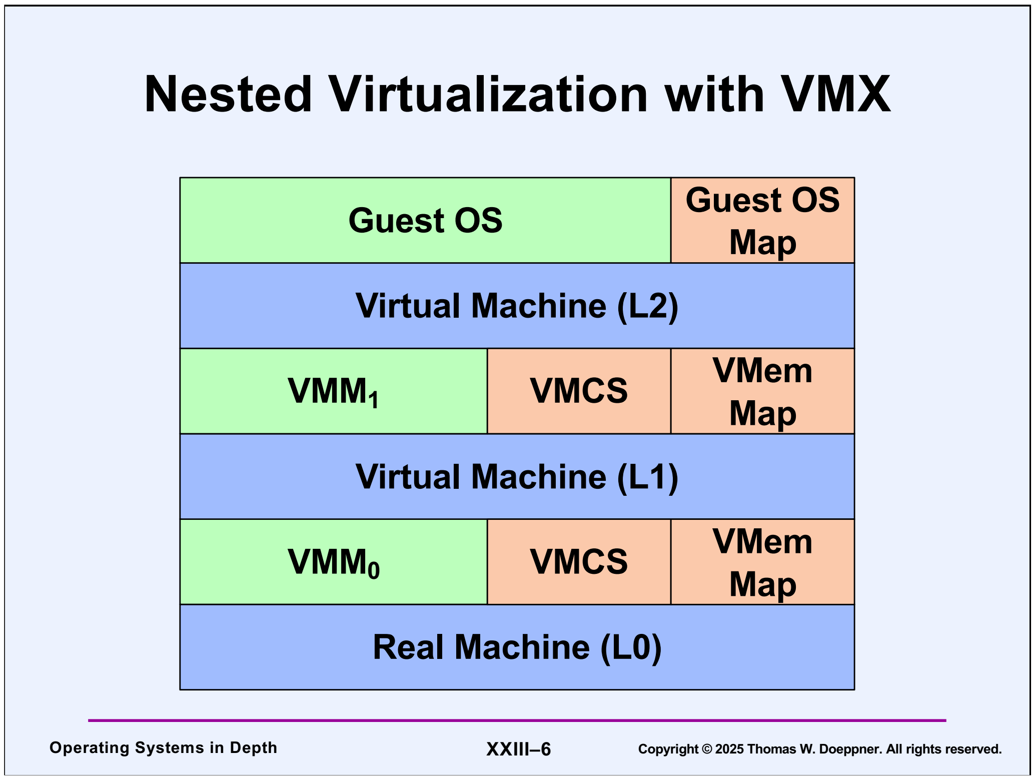


Nested Virtualization on VMX

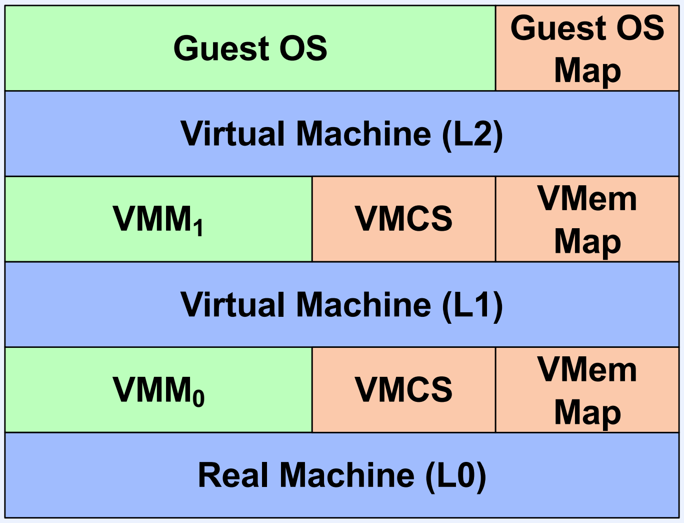
- A VMM is designed to use VMX extensions (including EPT)
- It supports VMs that appear to be real x86's (but without VMX extensions)
- Can the VMM run in a VM of the level-0 VMM?

Operating Systems in Depth XXIII–5 Copyright © 2025 Thomas W. Doeppner. All rights reserved.

The answer is no, at least not without some extra work, which we'll describe. The VMM requires the VMX extensions, which don’t exist at level 1.



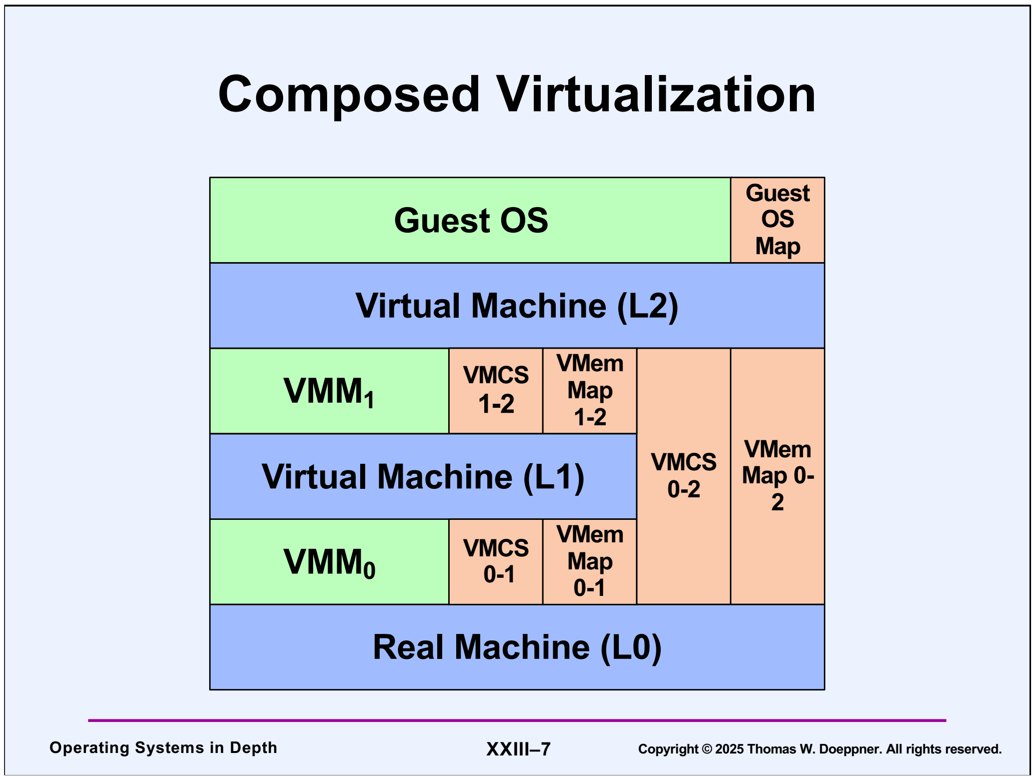
Nested Virtualization with VMX



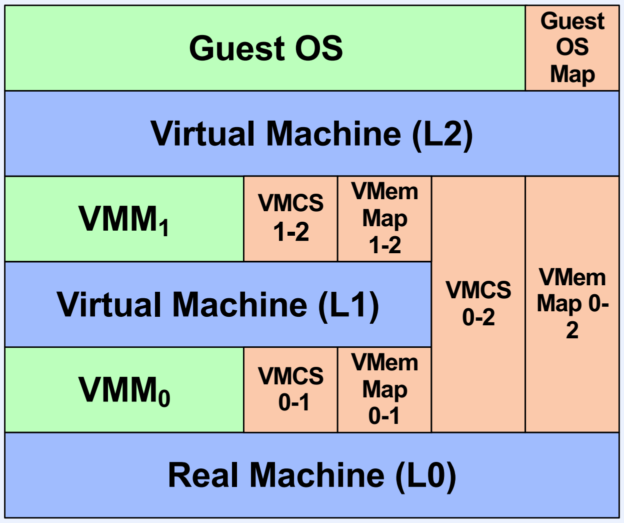
The slide illustrates what would happen in a "straightforward" approach to doing nested virtualization by virtualizing the VMX extensions. We have a guest OS running in a virtual machine supported by a VMM at level 2, which in turn is running in a virtual machine supported by a VMM at level 1, which is running in ring -1 on the real hardware. If the guest OS performs an operation causing a VMexit, control goes to VMM$_0$ running on the real machine in (real) ring -1. VMM$_0$ then must emulate the effect of a VMexit on the virtual machine at level 1 -- thus VMM$_1$ behaves as if it just received a VMexit. It in turn emulates the effect of a VMexit on the virtual machine level 2, Finally, that virtual machine can emulate whatever it was that its guest OS wanted to do in the first place.

But when VMM$_1$ does a VMenter to enter the guest OS, yet another VMexit occurs (down to VMM$_0$), and, again, this has to be propagated up to the Guest OS.

Also, note that each VMM must maintain a composite mapping of the virtual address translation for the levels above it.



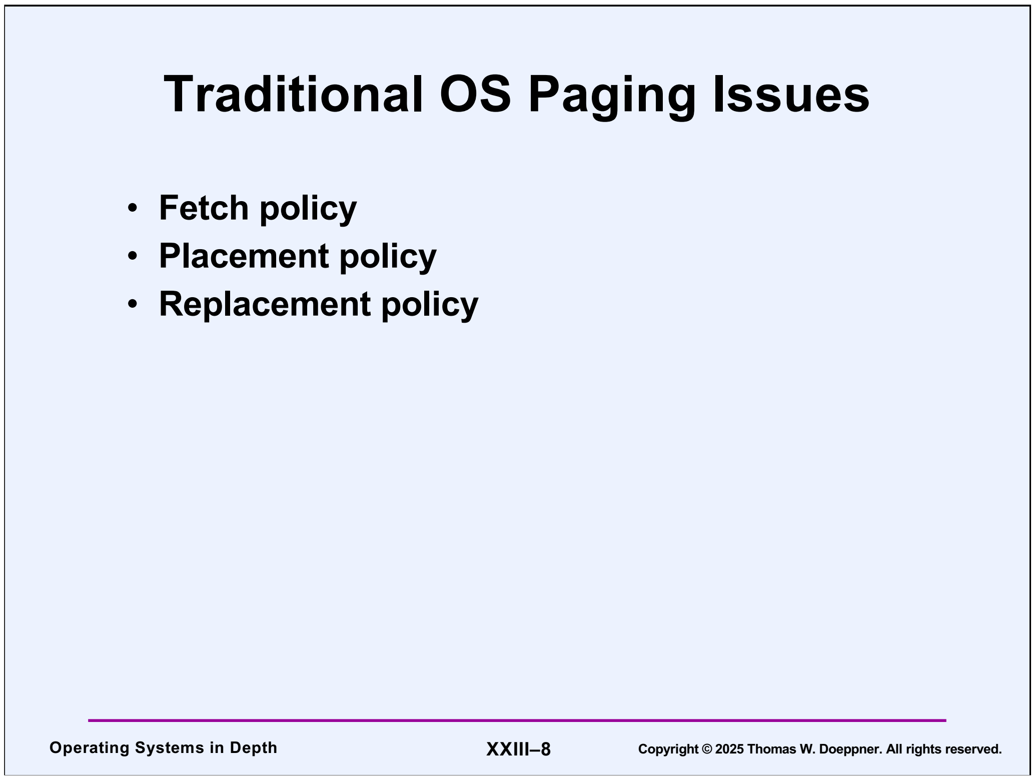
Composed Virtualization



A streamlined approach was taken in the aforementioned Turtles project. When a VMexit occurs in an upper VM, it, of course, must be handled by the bottom VMM. But the bottom VMM can, with some extra bookkeeping, give control to the upper VMM without having to involve the intermediate VMMs. Furthermore, it can set up a composite mapping from the upper VM's page table to real addresses that, in conjunction with EPT, can map the top address space to real memory.

The additional bookkeeping is in the form of virtual machine control structures (VMCSs) and composite memory maps. By keeping track of VMexits and VMenters, a VMM can determine the height of nested VMs above it. With that knowledge, it can maintain VMCSs for each of the nested VMs above it, so it can switch directly to one, bypassing the intermediate VMMs.

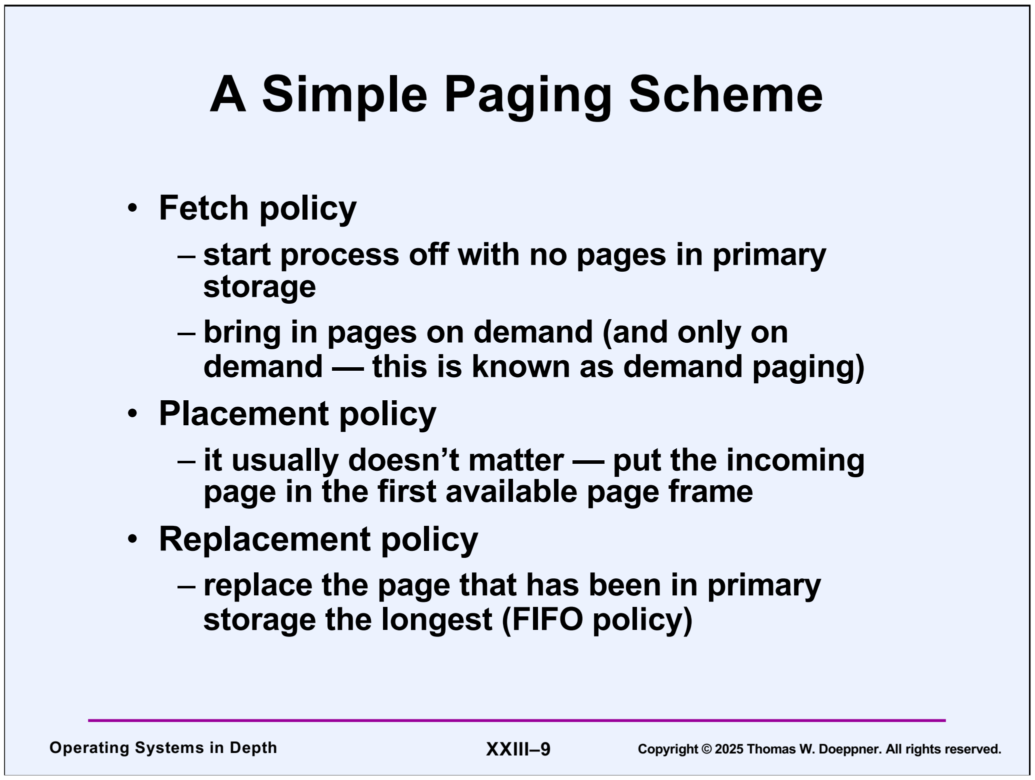
In the example shown in the slide, VMM_1 maintains $VMCS_{1-2}$ to represent the L2 virtual machine, while VMM_0 maintains both $VMCS_{0-1}$ to represent the L1 virtual machine and $VMCS_{0-2}$ to represent the L2 virtual machine. When VMM_1 attempts to do a VMenter with $VMCS_{1-2}$, a (real) VMexit occurs and control goes to VMM_0, which, in response, does a VMenter with $VMCS_{0-2}$. VMM_1 uses EPT to establish the composite mapping of the Guest OS Map and its mapping of the L2 virtual machine to the L1 virtual machine (which VMM_1 thinks is the real machine). However, this causes a VMexit to VMM_0, which uses EPT to create a composite mapping from the L2 virtual machine to the real machine (L0). Thus processes of the guest OS are running in an address space mapped by the guest-OS map to the L2 virtual machine, which is in turn mapped to the L0 address space by VMM_0's EPT mapping.



Traditional OS Paging Issues

- Fetch policy
- Placement policy
- Replacement policy

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).



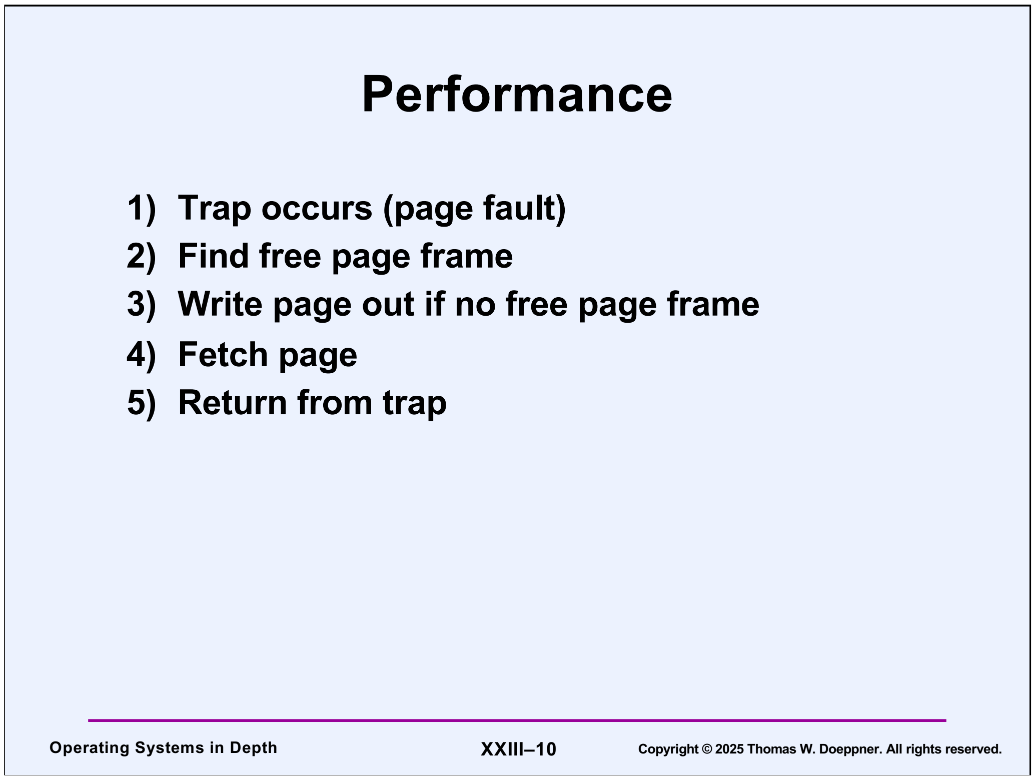
A Simple Paging Scheme

- 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 usually 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.)

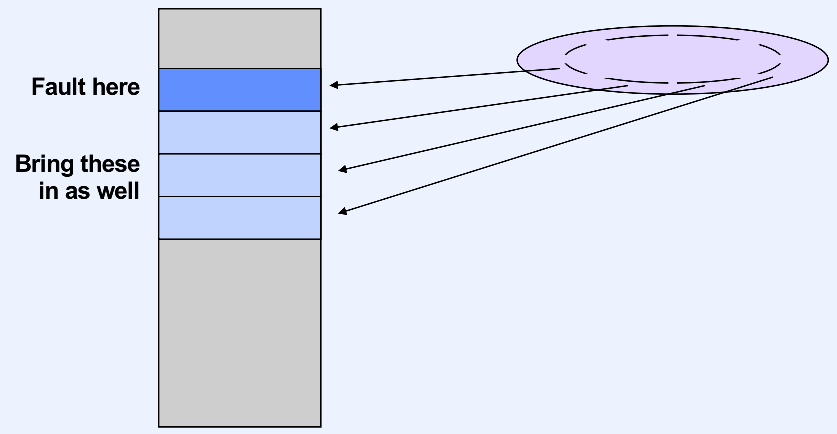


Performance

1) Trap occurs (page fault)
2) Find free page frame
3) Write page out if no free page frame
4) Fetch page
5) Return from trap

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.

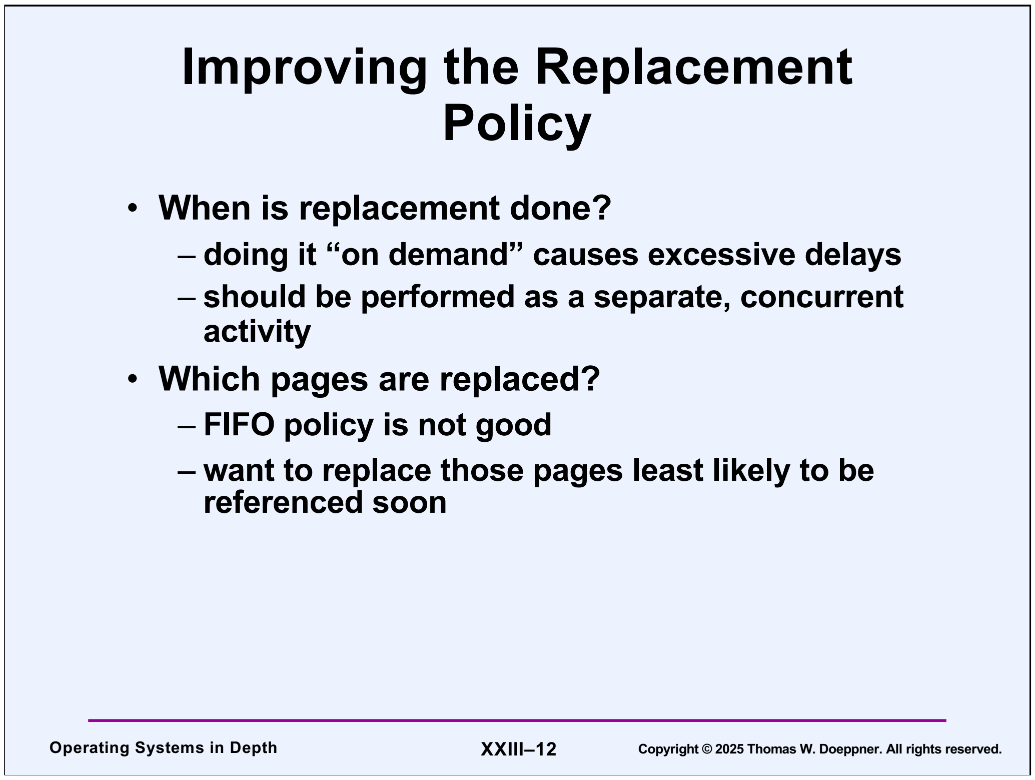
Improving the Fetch Policy



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.



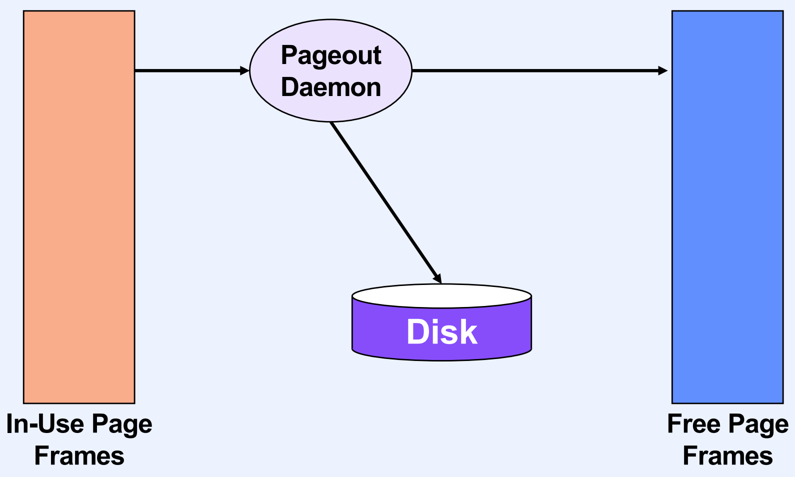
Improving the Replacement Policy

- 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 leaves 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.

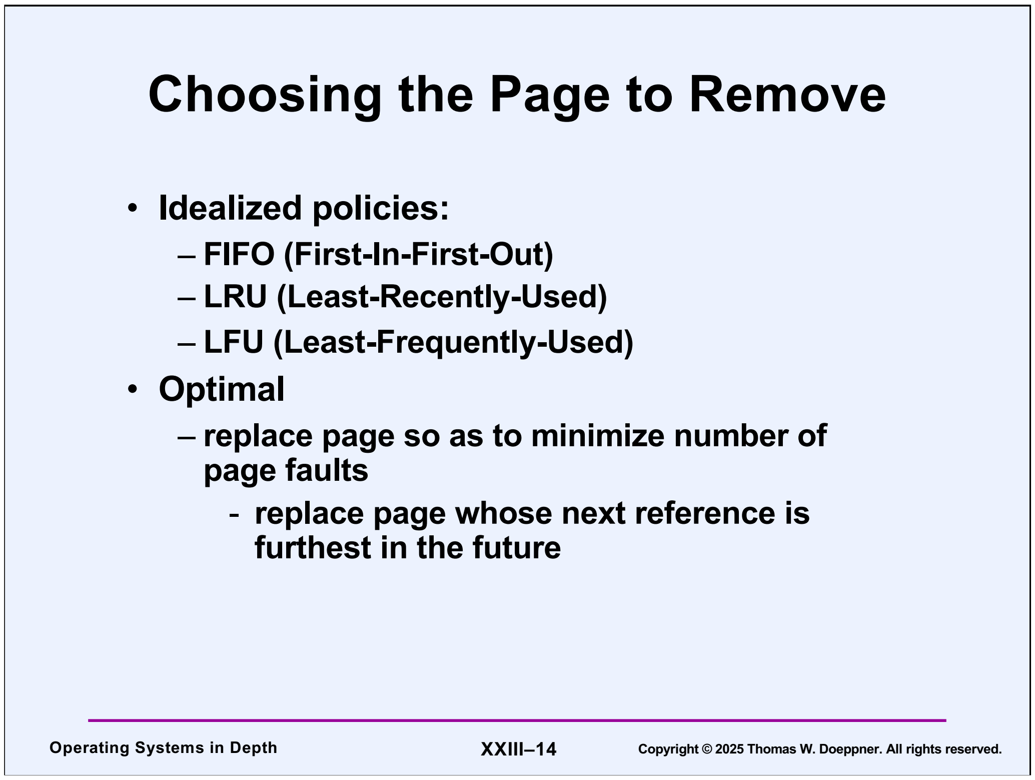
The other problem with the replacement policy is the use of the FIFO technique for deciding which pages to remove from primary storage—the page that has been in memory the longest could well be the page that is getting the vast majority of the references. Ideally, we would like to remove from primary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

The “Pageout Daemon”



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.



Choosing the Page to Remove

- Idealized policies:
  - FIFO (First-In-First-Out)
  - LRU (Least-Recently-Used)
  - LFU (Least-Frequently-Used)
- Optimal
  - replace page so as to minimize number of page faults
    - replace page whose next reference is furthest in the future

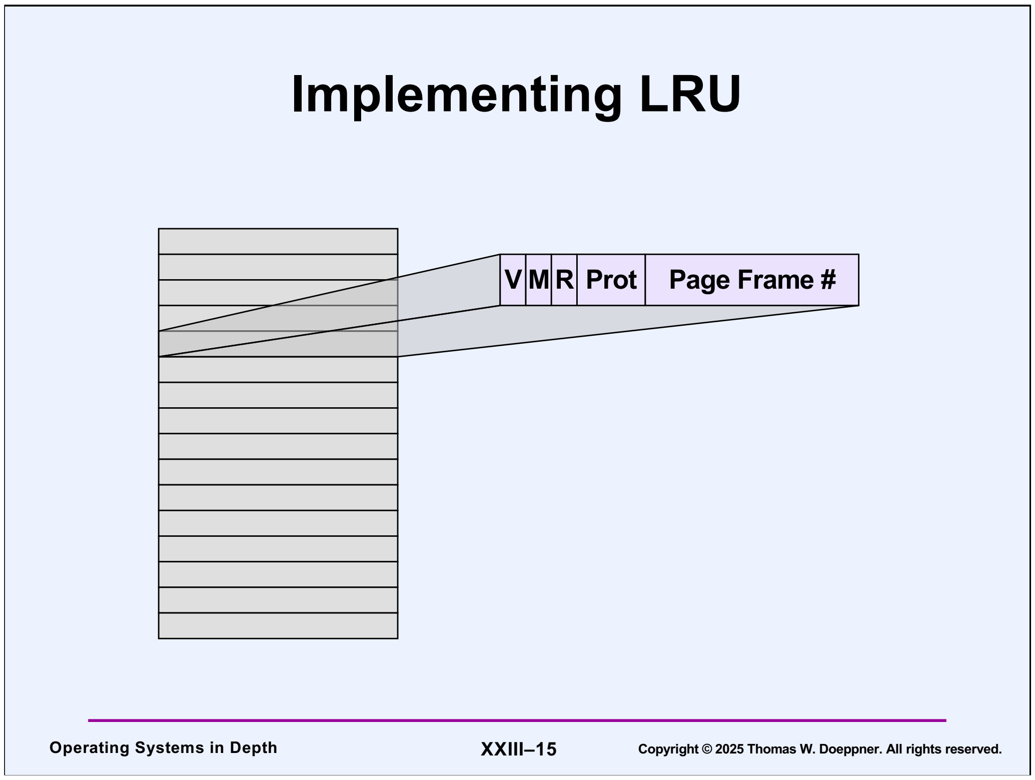
Determining which pages to replace is a decision that could have significant ramifications to system performance. If the wrong pages are replaced, i.e. pages which are referenced again fairly soon, then not only must these pages be fetched again, but a new set of pages must be tagged for removal.

The FIFO scheme has the benefit of being easy to implement, but often removes the wrong pages. 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.

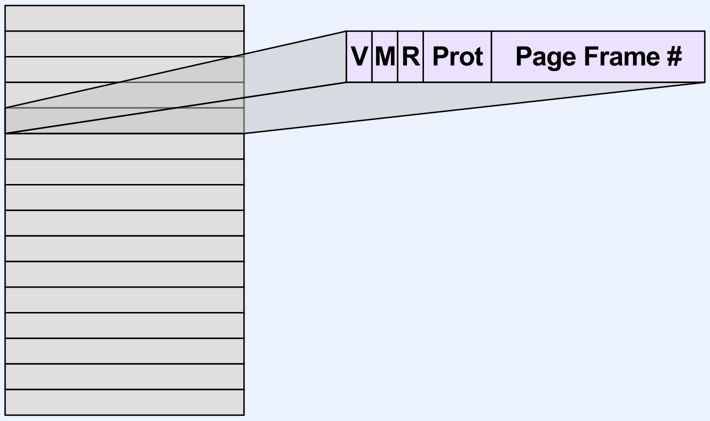
A variant of this approach is LFU—least frequently used. Pages are ordered by their frequency of use, and we replace the page which has been used the least frequently of those still in primary storage.

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. It is straightforward to implement LRU for the Unix buffer cache, since the time to order the buffers is insignificant compared to the time required to access them. But we cannot afford to maintain pages in order of last reference.

The optimal policy would, of course, be to replace pages so as to minimize the number of page faults. It was proven by Laszlo Belady in the 1960s that one can accomplish this by replacing the page whose next reference is the furthest in the future, but this, of course, requires compete knowledge of the future execution of the program.



Implementing LRU



The standard approach for implementing the LRU (and LFU) strategy is to approximate it. We divide time into “epochs” and order pages by whether 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.

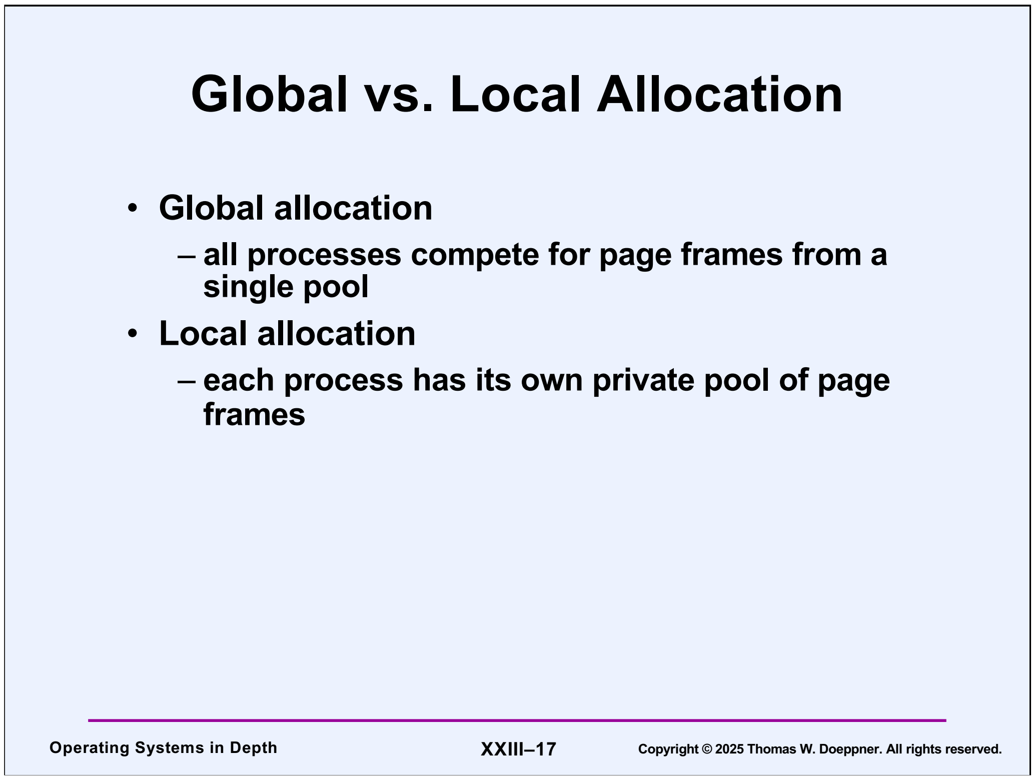
Quiz 1

Your computer is running one process. Pretty much all available real memory is being actively used and processor utilization is around 90%. You now add another process that's similar to the first in terms of both memory and processor utilization (though it's running a different program). Assume the LRU page replacement policy is used.

a) Processor utilization will rise to nearly 100%

b) Processor utilization will stay at around 90%

c) Processor utilization will drop precipitously

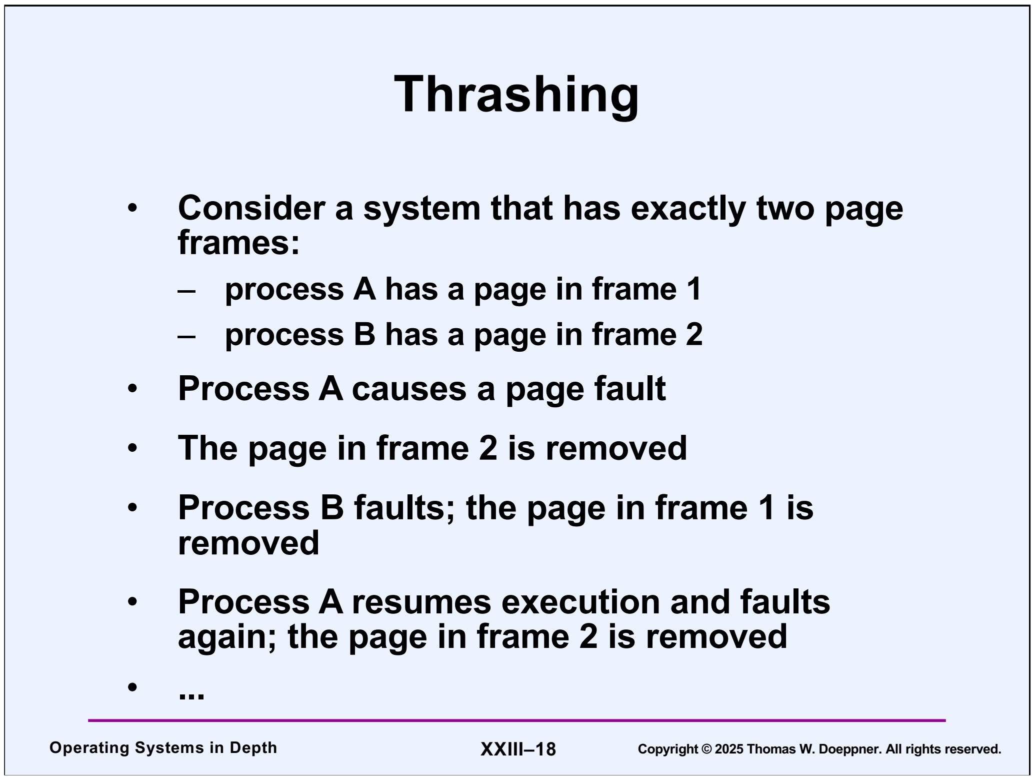


Global vs. Local Allocation

- Global allocation
  - all processes compete for page frames from a single pool
- Local allocation
  - each process has its own private pool of page frames

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 we have to determine how many page frames to give to each process.

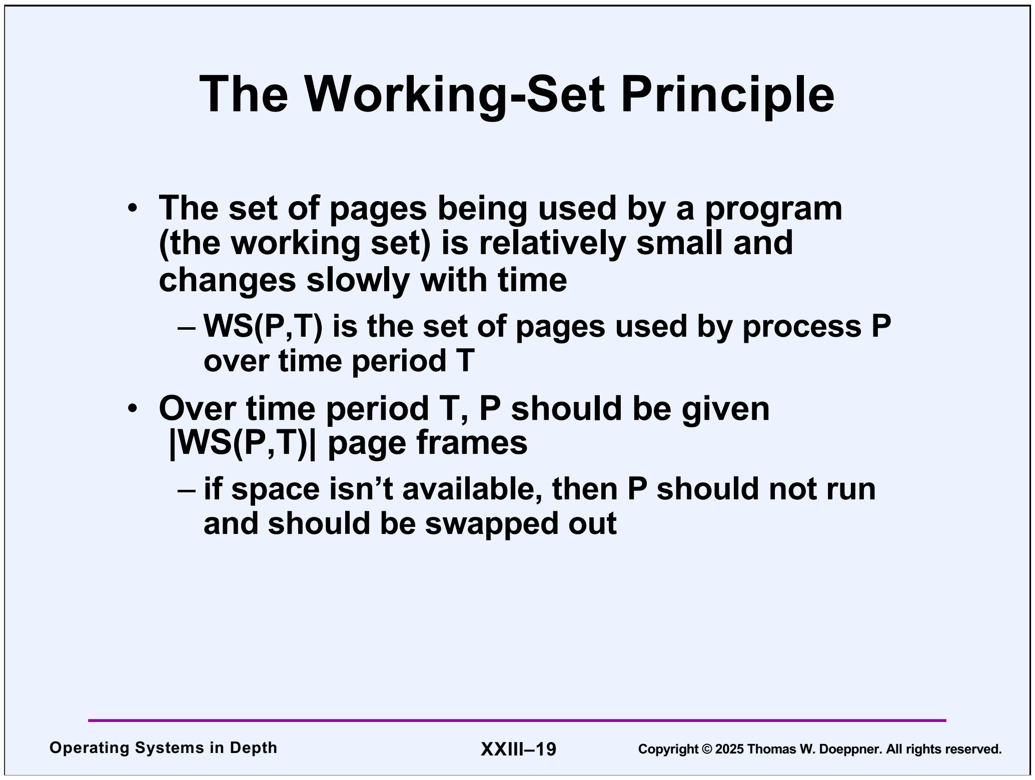


Thrashing

- Consider a system that has exactly two page frames:
  - process A has a page in frame 1
  - process B has a page in frame 2
- Process A causes a page fault
- The page in frame 2 is removed
- Process B faults; the page in frame 1 is removed
- Process A resumes execution and faults again; the page in frame 2 is removed
- ...

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).



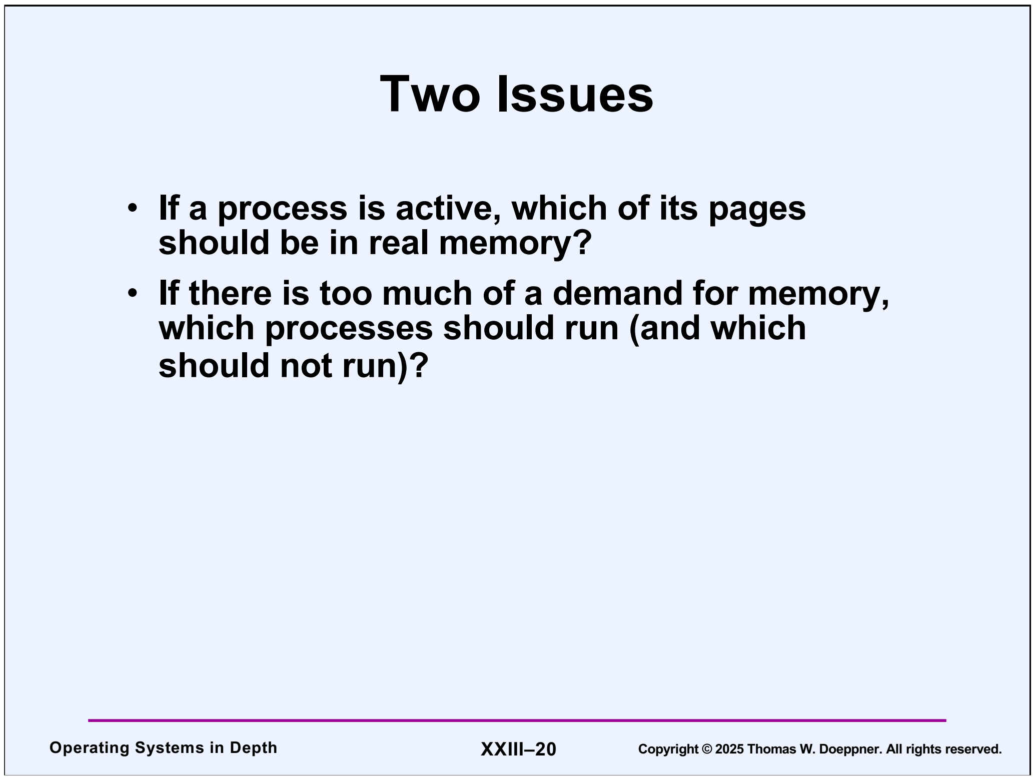
The Working-Set Principle

- The set of pages being used by a program (the working set) is relatively small and changes slowly with time
  - WS(P,T) is the set of pages used by process P over time period T
- Over time period T, P should be given |WS(P,T)| page frames
  - if space isn't available, then P should not run and should be swapped out

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 ensure 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.

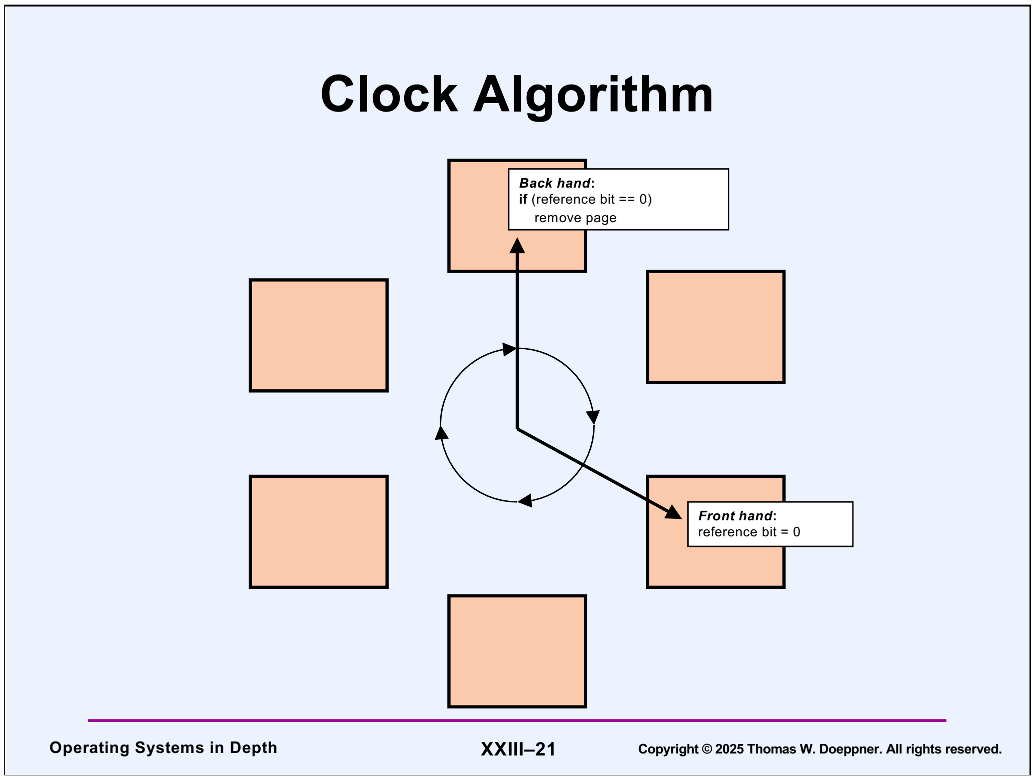
The working-set principle is used in few, if any systems (the size of the working set is just too nebulous a concept). However, it is very important as an ideal to which paging systems are compared.



Two Issues

- If a process is active, which of its pages should be in real memory?
- If there is too much of a demand for memory, which processes should run (and which should not run)?

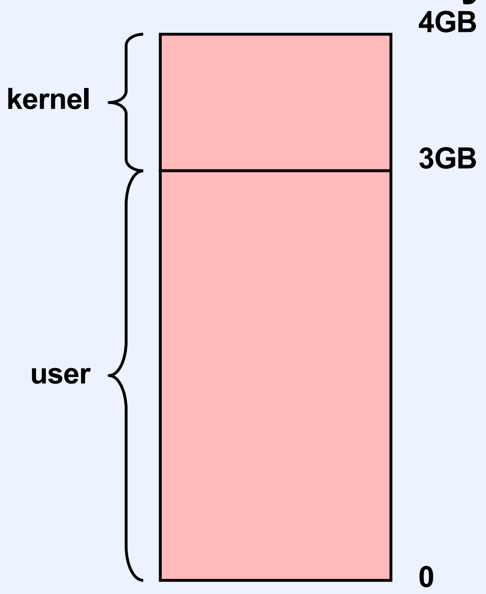
Much more attention is paid to the first issue than to the second issue.



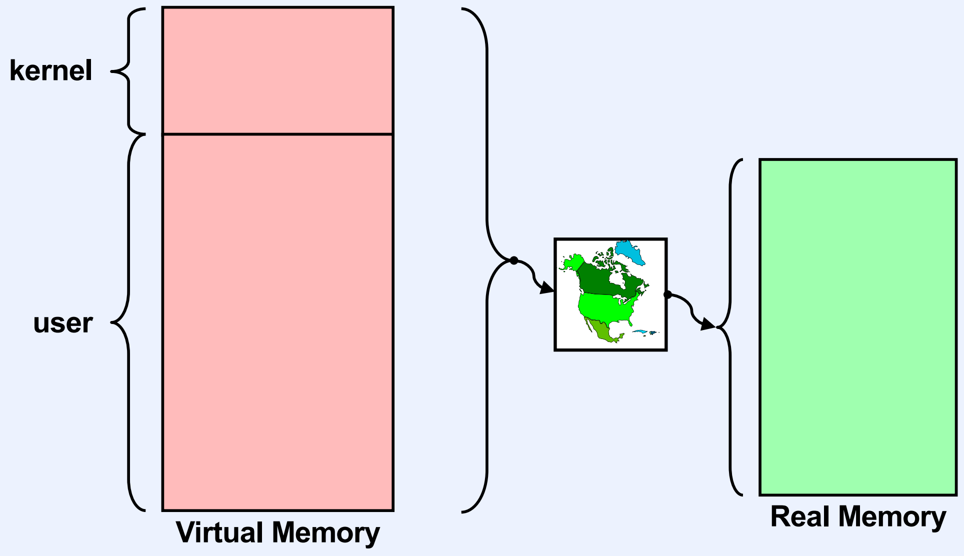
Clock Algorithm

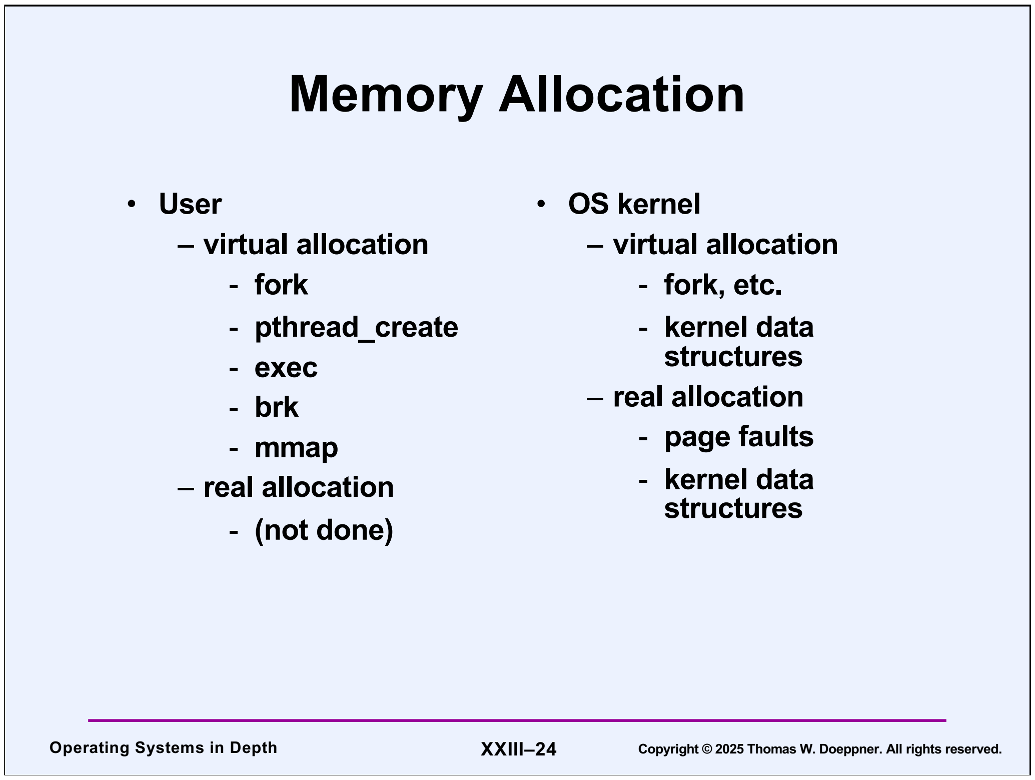
The clock algorithm is an approximation of LRU combined with LFU. As long as a page is referenced sufficiently often, it stays in memory. A primary benefit of the clock algorithm is that its easily implemented and doesn’t have a high cost in terms of processor time.

Linux Intel x86 VM Layout



Real Memory

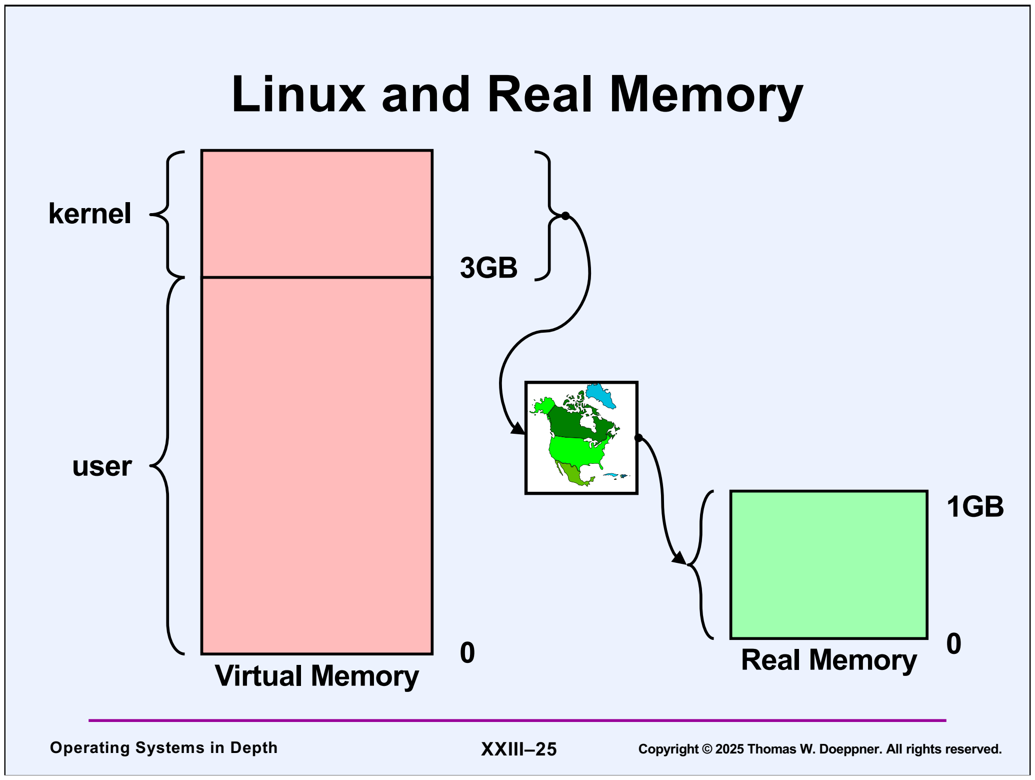




Memory Allocation

- User
  - virtual allocation
    - fork
    - pthread_create
    - exec
    - brk
    - mmap
  - real allocation
    - (not done)
- OS kernel
  - virtual allocation
    - fork, etc.
    - kernel data structures
  - real allocation
    - page faults
    - kernel data structures

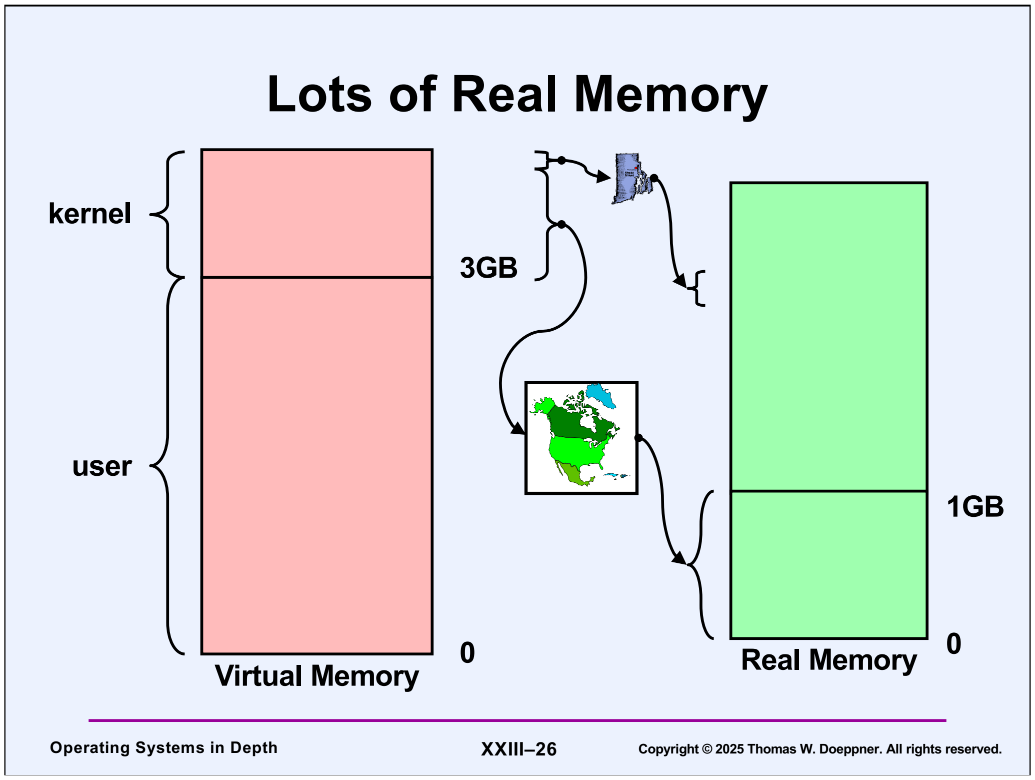
Note that in Linux, as well as in Weenix, there are no page faults when accessing kernel memory. Almost all kernel pages are permanently mapped to real memory. For the few that aren't, the kernel makes sure they are mapped before they are referenced.



Linux and Real Memory

Linux, like most operating systems, doesn't separately allocate virtual memory and real memory for its own data structures—it allocates the two together. But it also must allocate real memory to user processes separately from virtual memory. The "trick" it employs is to map all of real memory into its portion of the address space. Thus it can manage kernel virtual memory and all of real memory at the same time. Some of the real memory, that backing up kernel text and data, is dedicated to the kernel and is not reused for other things. The rest is used to satisfy both user and kernel memory requests. So, the kernel can allocate virtual memory for its own data structures and get real memory as well. When the kernel allocates real memory for user processes, and thus maps that real memory into a user address space, the real memory becomes "double-mapped"—it's mapped both into the kernel address space and the user address space.

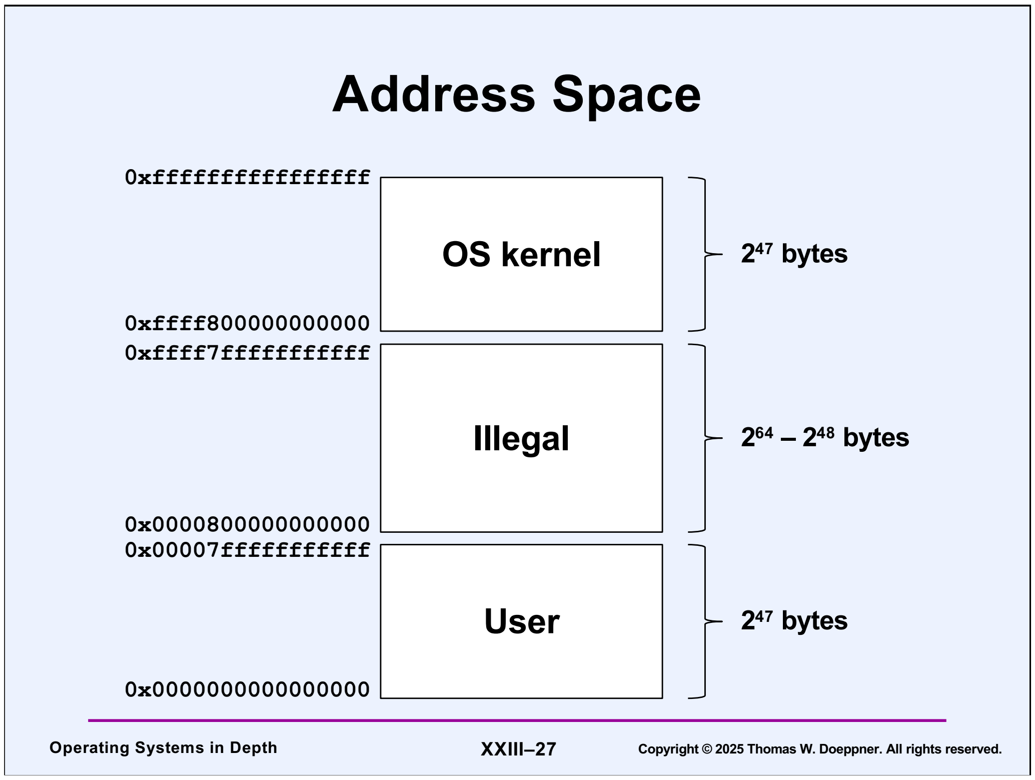
For the x86, the kernel sets up that portion of its address space that is not shared with user processes so that almost all of it, from PAGE_OFFSET (normally 3GB) to 4GB–128MB (i.e., all but the last 128MB of the address space—why this is needed is described in the next slide) maps to the first one gigabyte of real (physical) memory. Thus the kernel itself resides completely in real memory. That portion of real memory that it doesn't need is made available to user processes, though note that memory that is mapped by user processes is also mapped by the kernel.



Lots of Real Memory

If a system has more than one gigabyte of real memory, there's a problem—not all of it can be mapped into the kernel. What's done is to map the first 896MB (1024-128) into the first 896MB of the kernel address space, reserving a portion of the address space (known as the "kmap region") for dynamically mapping additional real memory into the kernel whenever the kernel needs to reference it. This means that real memory for kernel data structures must come from the first 896MB, since the kmap region is reserved for temporary mappings.

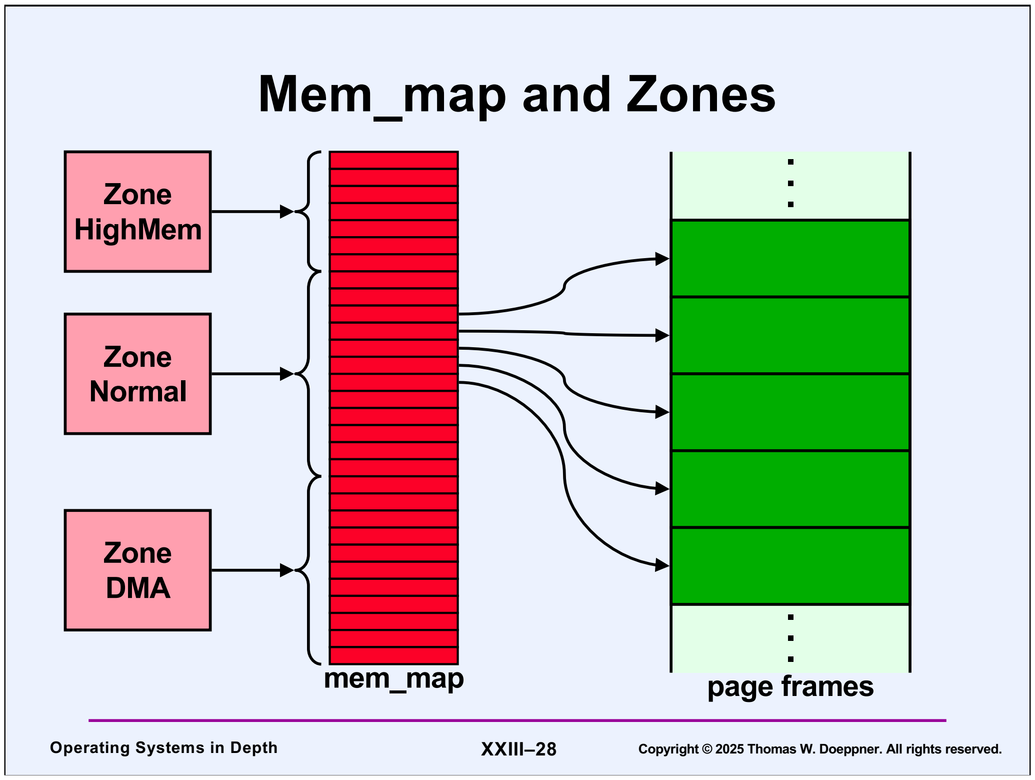
Note that the kernel has direct access to the pages of the current process, since they are in the lower ¾ of the address space. The kmap region must be set up in situations in which, for example, the kernel must reference pages that are not in the current process's address space. This might happen when a device driver must access memory that's not mapped into the current process's address space, but is mapped into another process's address space.



Address Space

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.

With a 64-bit address space and current memory prices, it becomes straightforward (again) for the kernel to map all of real memory into its address space.



Mem_map and Zones

In Linux, each page frame is represented by an entry in the **mem_map** array. These entries represent the state of that page frame, showing what is mapped into it, what queues it is on, etc.

Zone **DMA** is that portion of physical memory that can be used for DMA transfers by ISA bus devices (as well as for all other purposes). On x86 PCs, it covers memory addresses up to 16MB. Zone **Normal** are those page frames, beyond zone DMA, that can be used for both kernel and user use. It covers memory addresses beyond 16MB up to 896MB. Finally, zone **HighMem** are those page frames that can be used in user applications, but cannot be permanently mapped into the kernel. It covers memory addresses beyond 896MB.

On 64-bit implementations (e.g., for the x86-64), there is also a zone DMA32 in which memory is allocated for devices whose controllers use 32-bit addresses, but not 64-bit addresses. Thus it covers memory addresses up to 4GB.

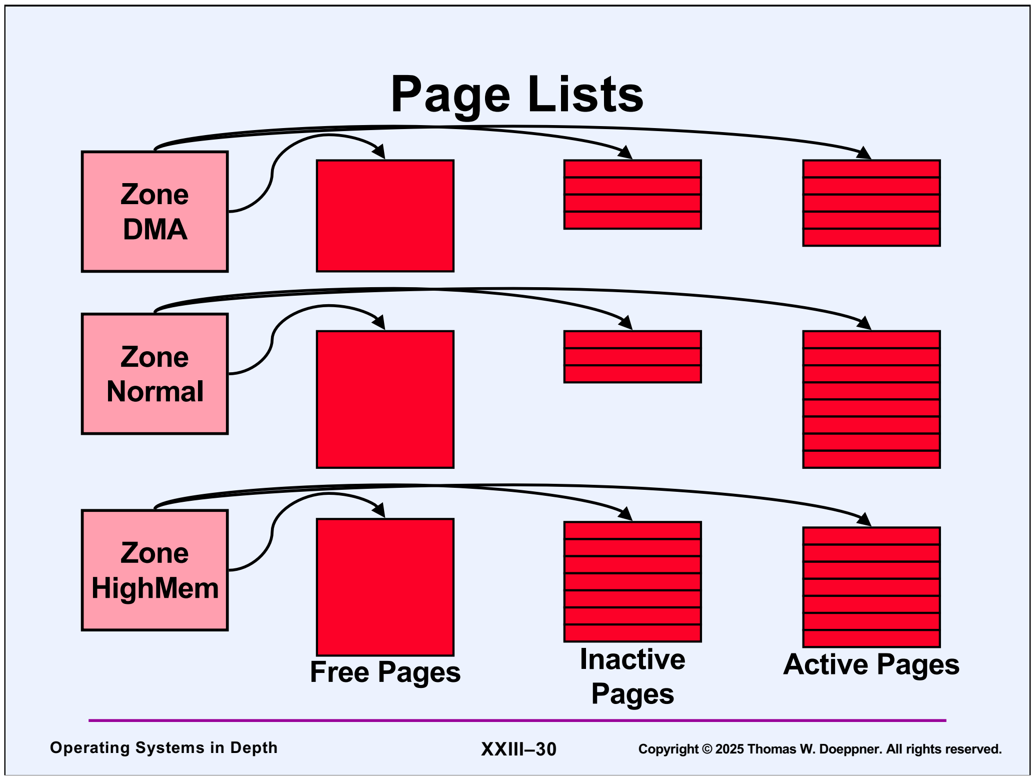
Quiz 2

We have a disk whose controller uses 64-bit memory addresses. We’d like to set it up for a read that will transfer 32K bytes. Thus the block will be read into a buffer that occupies eight 4KB pages.

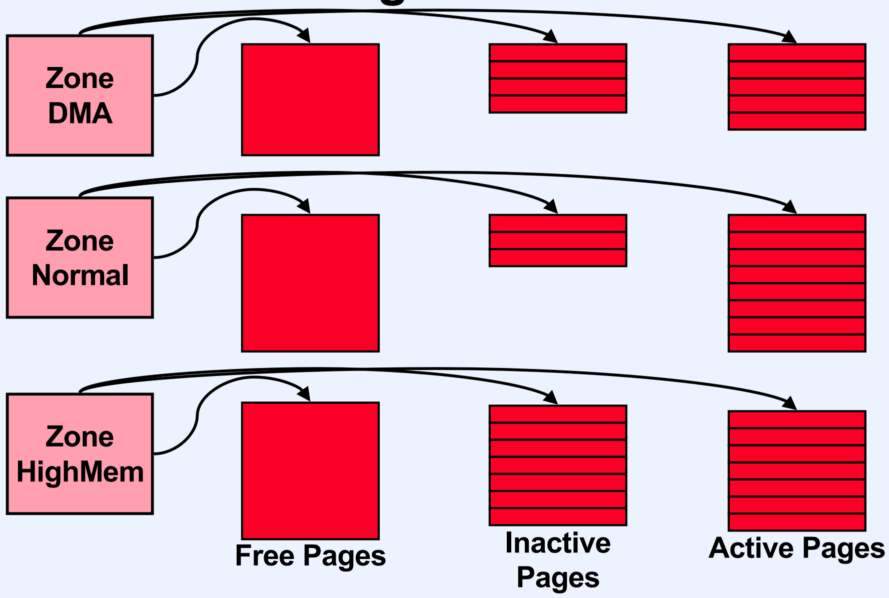
a) Since our system uses virtual memory, the buffer occupies contiguous pages of virtual memory, which may be mapped into non-contiguous page frames of real memory

b) As in a, except that the contiguous pages of virtual memory need not have valid mappings (and thus page faults are generated and handled)

c) The buffer must occupy eight page frames of contiguous real memory



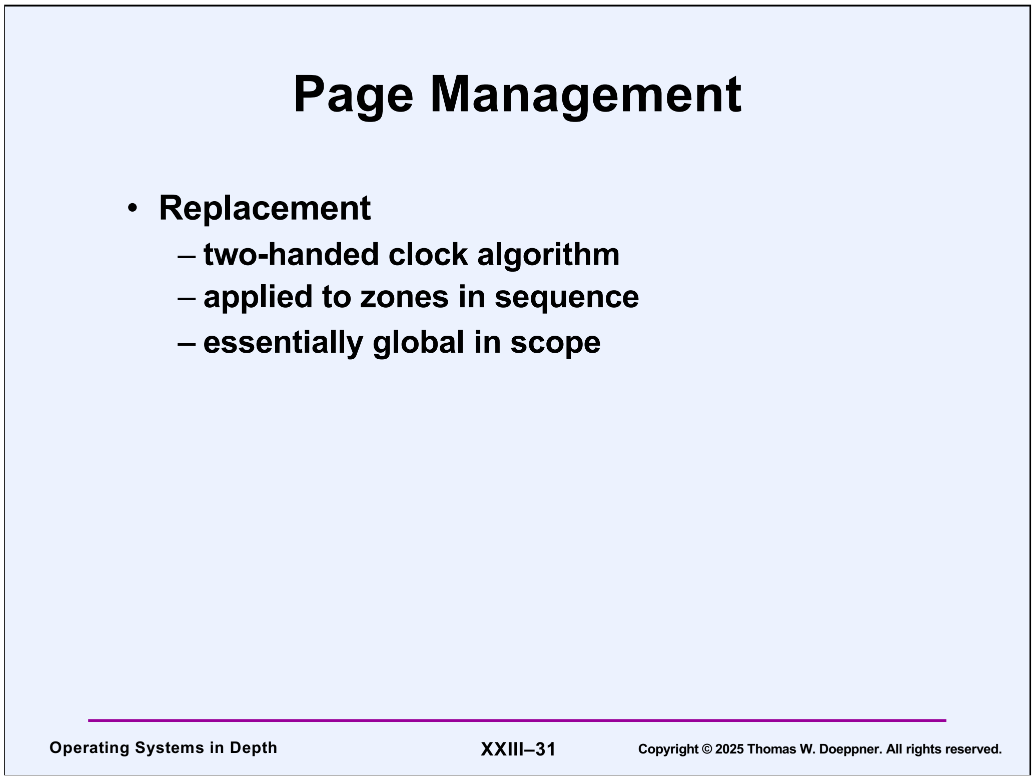
Page Lists



Associated with each zone are three lists of page frames: those page frames from each zone that are considered to be in active use are in the zone's list of **active pages**. Those that have not been used recently are put into the zone's list of **inactive pages**. Finally, those page frames that are the best candidates for new use are put in the zone's list of **free pages**. We'll discuss soon the mechanism used for deciding when to move page frames from list to list.

Page frames used for kernel data structures are marked as such and are not on any of the lists. Page frames in the active and inactive dirty lists are mapped into process address spaces. However, page frames in the inactive clean lists are not mapped into process address spaces, though they still contain useful data and can be remapped if required. Page frames in the free lists are not considered to contain useful data and are definitely not mapped into process address spaces.

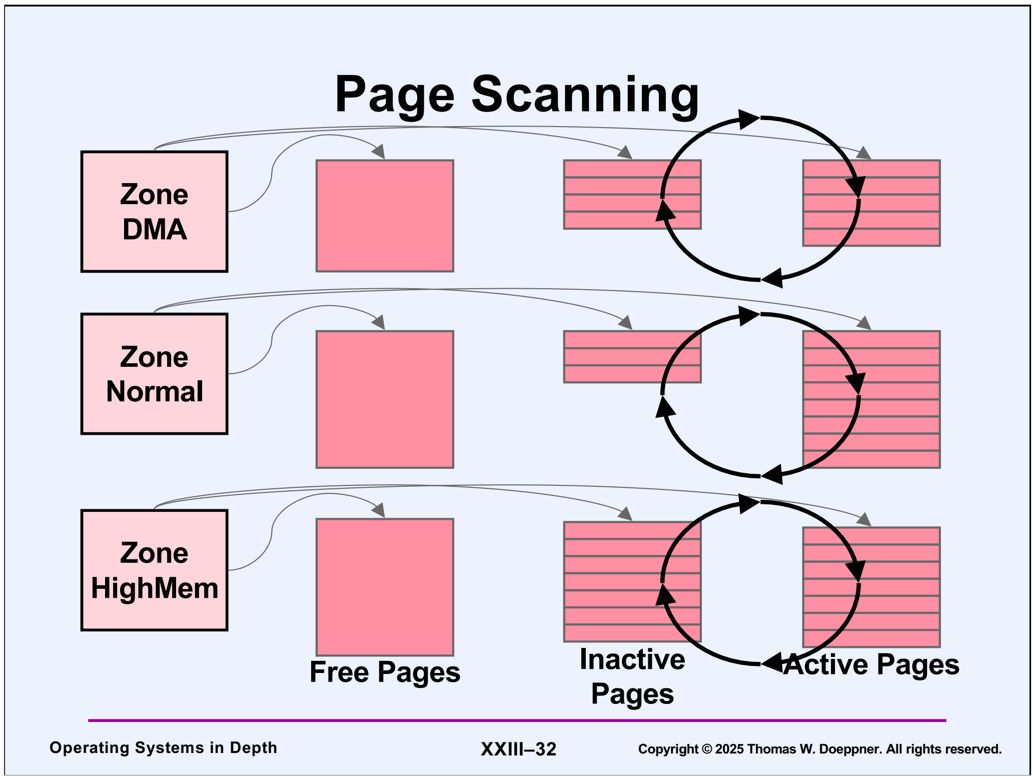
Note that the free page list is not a simple list of pages, but a buddy list, as to be described.



Page Management

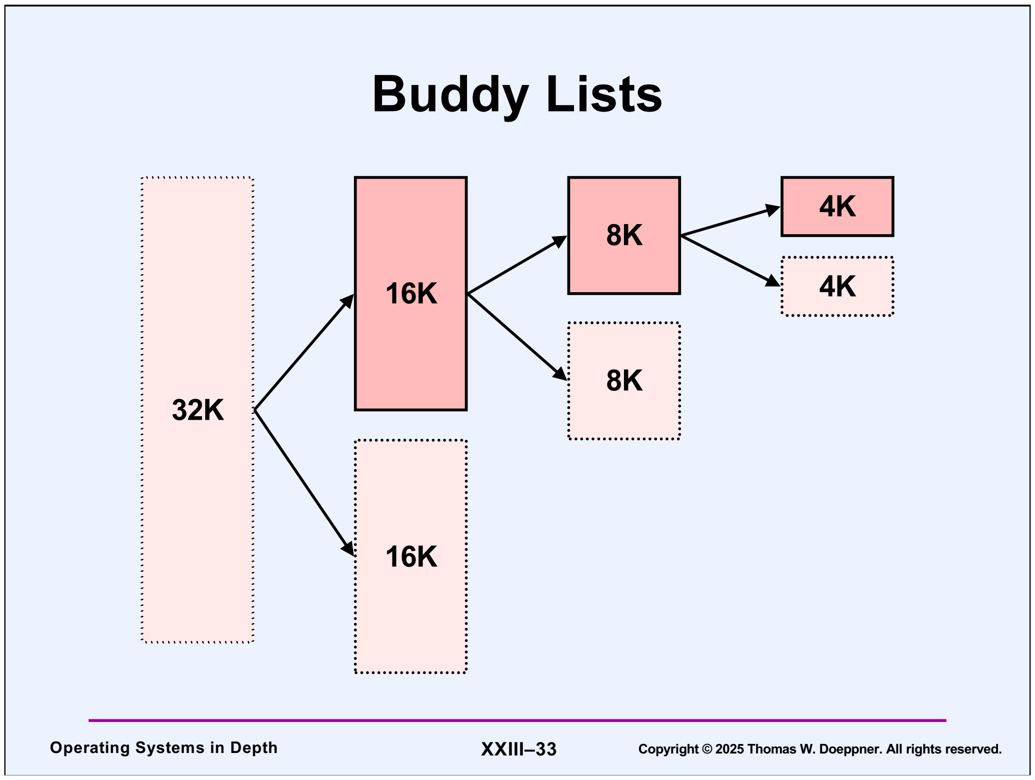
- Replacement
  - two-handed clock algorithm
  - applied to zones in sequence
  - essentially global in scope

We are continuing to discuss how pages are managed in Linux.

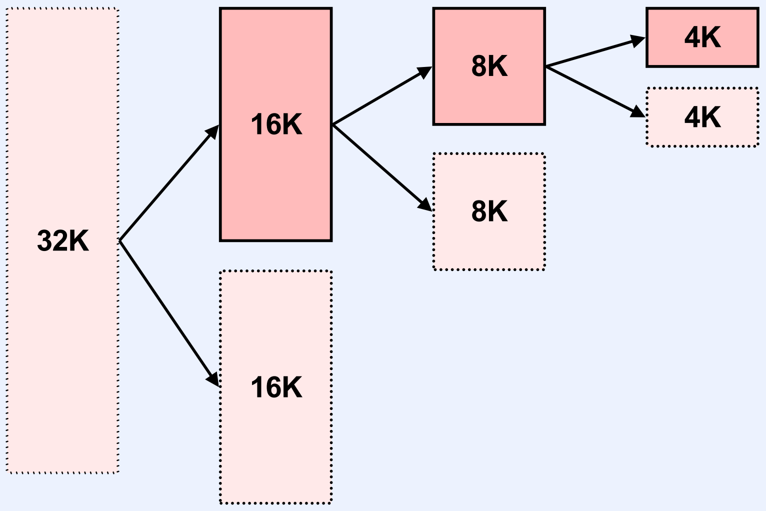


Page Scanning

A special kernel process, known as **kswapd**, scans pages in the active and inactive lists. Active pages found to be unreferenced are moved to the inactive list. Inactive pages found to be referenced are moved to the active list. Inactive pages found to be unreferenced are moved to the free list, after first being written out, if necessary. In addition, a number of other kernel processes, known as **pdflush** threads, periodically write out modified pages.



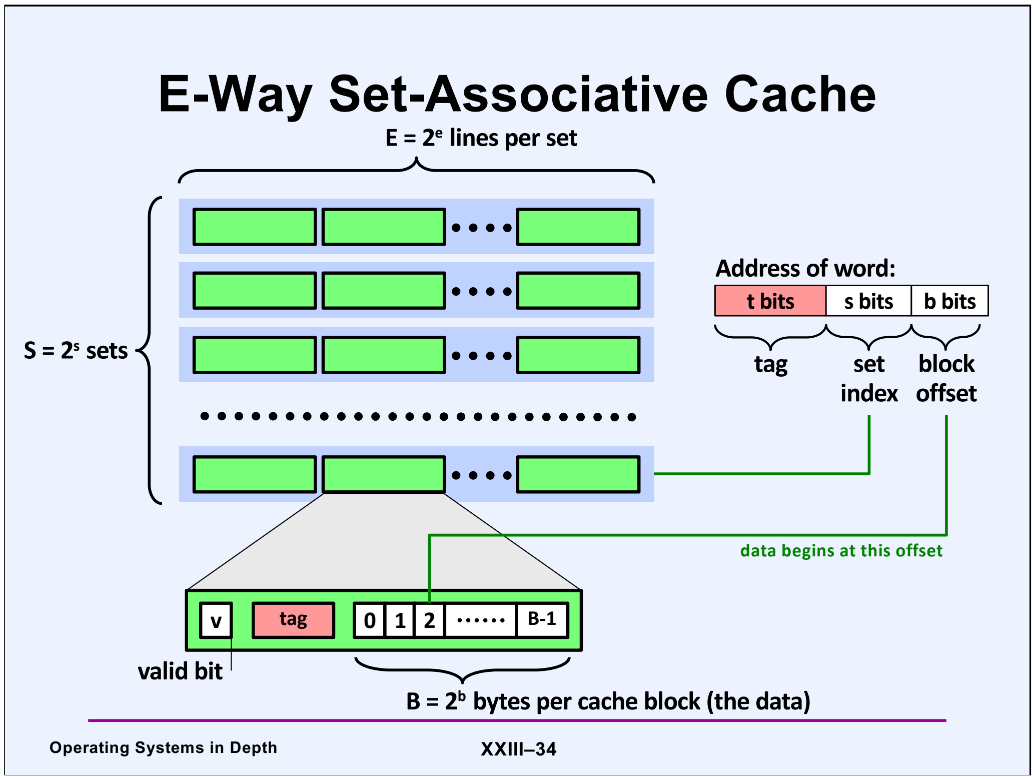
Buddy Lists



The buddy system is a simple dynamic storage allocation scheme that works surprisingly well. Storage is maintained in blocks whose sizes are powers of two. Requests are rounded up to the smallest such power greater than the request size. If a block of that size is free, it's taken, otherwise the smallest free block greater than the desired size is found and split in half—the two halves are called buddies. If the size of either buddy is what's needed, one of them is allocated (the other remains free). Otherwise, one of the buddies is split in half. This splitting continues until the appropriate size is reached.

Liberating a block now is easy: if the block's buddy is free, you join the block being liberated with its buddy, forming a larger free block. If this block's buddy is free, you join the two of them, and so forth until the largest free block possible is formed.

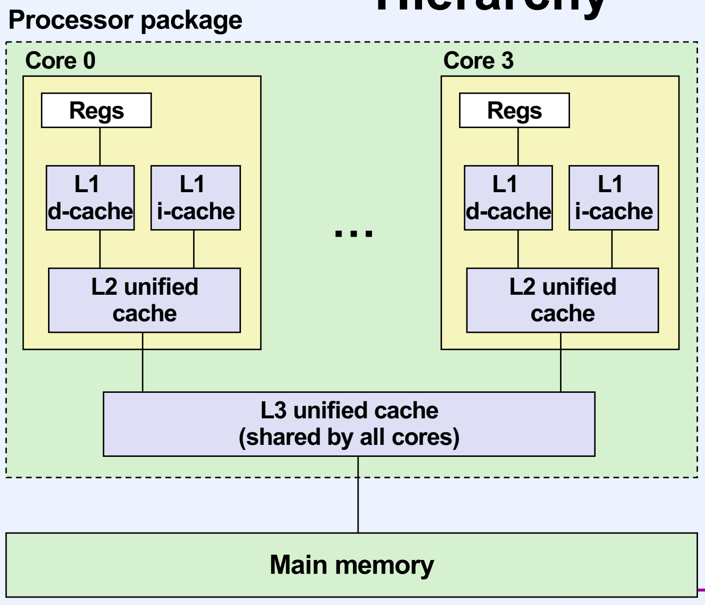
One bit in each block, say in its first or last byte, is used as a tag to indicate whether the block is free. Determining the address of a block's buddy is simple. If the block is of size 2^k, then the rightmost k-1 bits of its address are zeros. The next bit (to the left) is zero in one buddy and one in the other; so, you simply complement that bit to get the buddy's address.



E-Way Set-Associative Cache

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

Intel Core i5 and i7 Cache Hierarchy



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 11 cycles

L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for all caches

Supplied by CMU and also used in CS 33.

Quiz 3

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

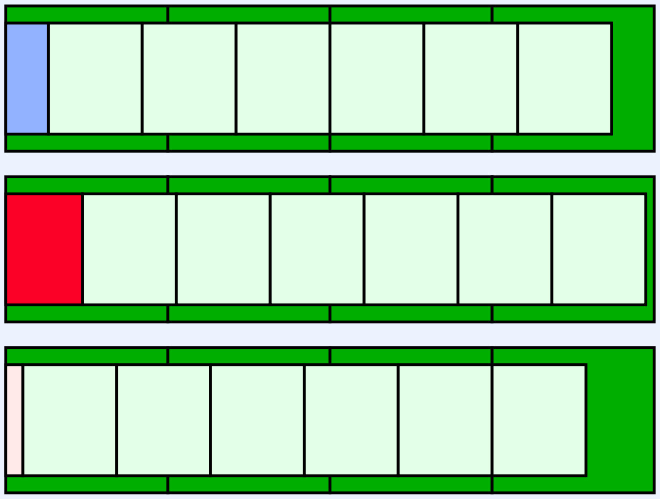
a) Rounding the size of the data structure up to a power of 2

b) Making sure all reside in the same cache set

c) Making sure they are distributed across cache sets

d) Nothing would help

Slab Allocation



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

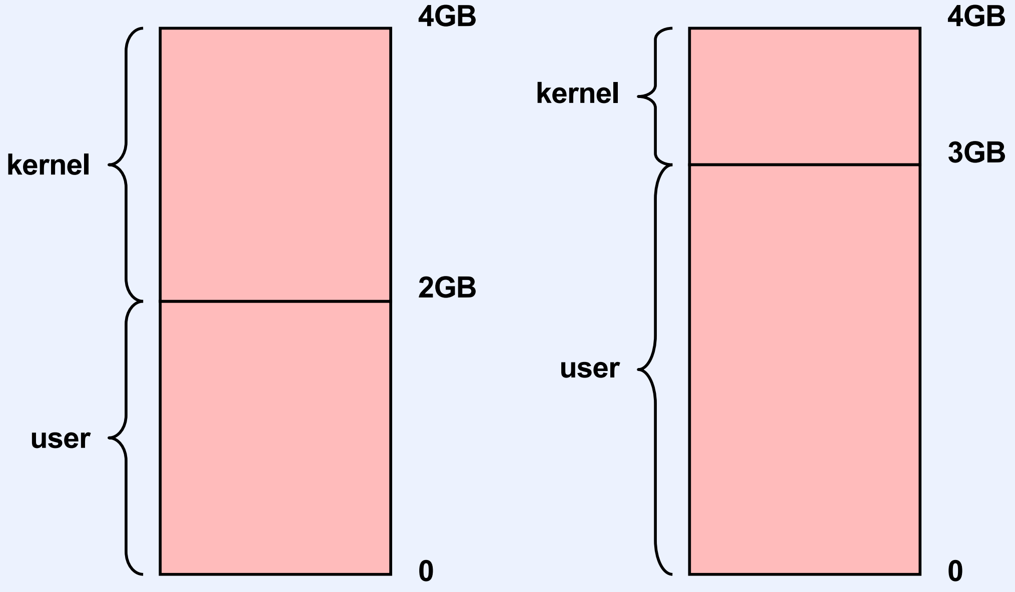
Slab allocation is a relatively new technique (developed at Sun: see "The Slab Allocator: An Object-Caching Kernel Memory Allocator," by Jeff Bonwick and appearing in the **Proceedings of the USENIX Summer 1994 Technical Conference**). For each type of object to be managed in such a fashion, a separate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

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

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

Slab allocation is used in Weenix.

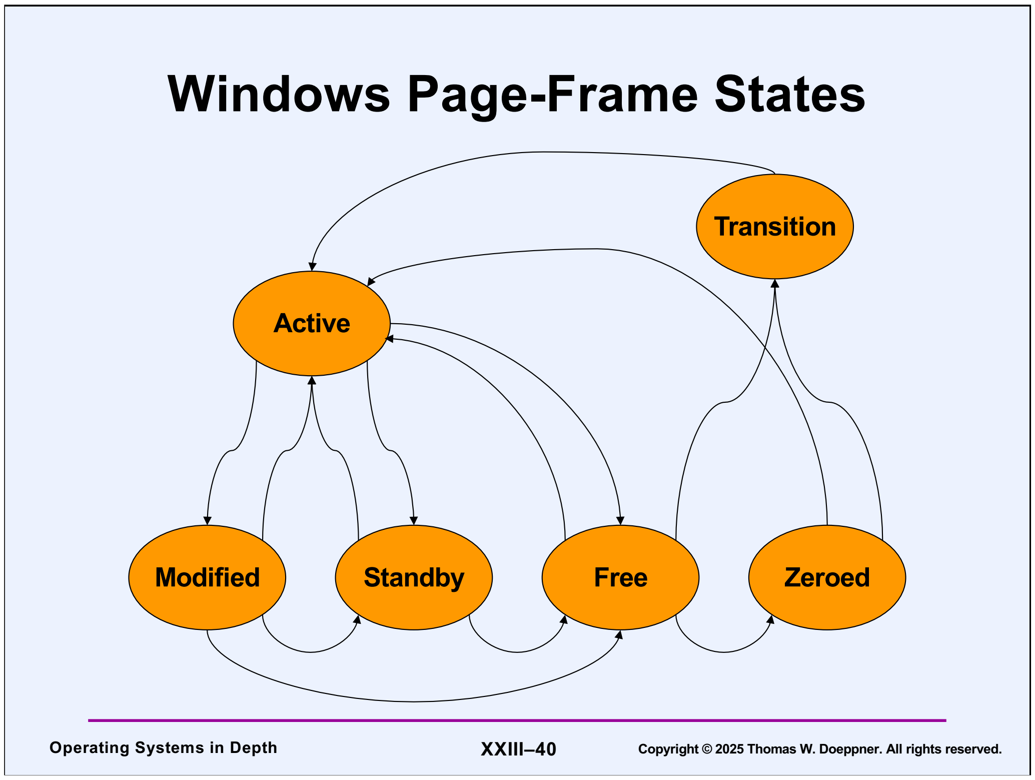
Windows x86 Layout



Either arrangement may be configured.

Windows Paging Strategy

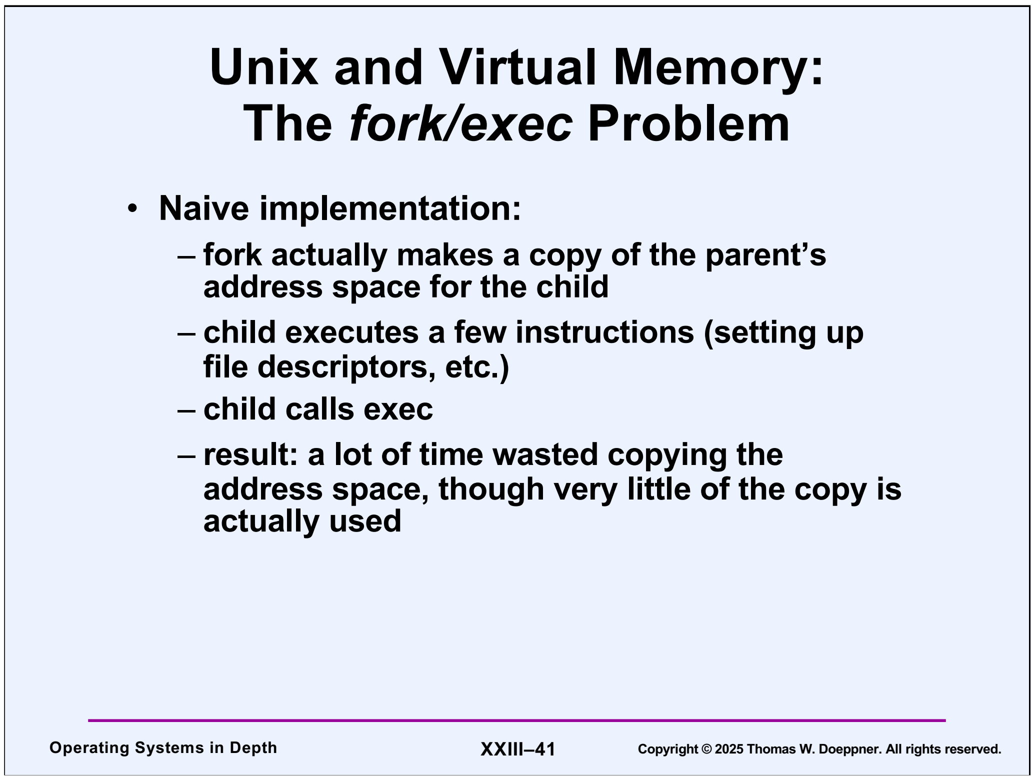
- All processes guaranteed a “working set”
  - lower bound on page frames
- Competition for additional page frames
- “Balance-set” manager thread maintains working sets
  - one-handed clock algorithm
- Swapper thread swaps out idle processes
  - first kernel stacks
  - then working set
- Some of kernel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



Windows Page-Frame States

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

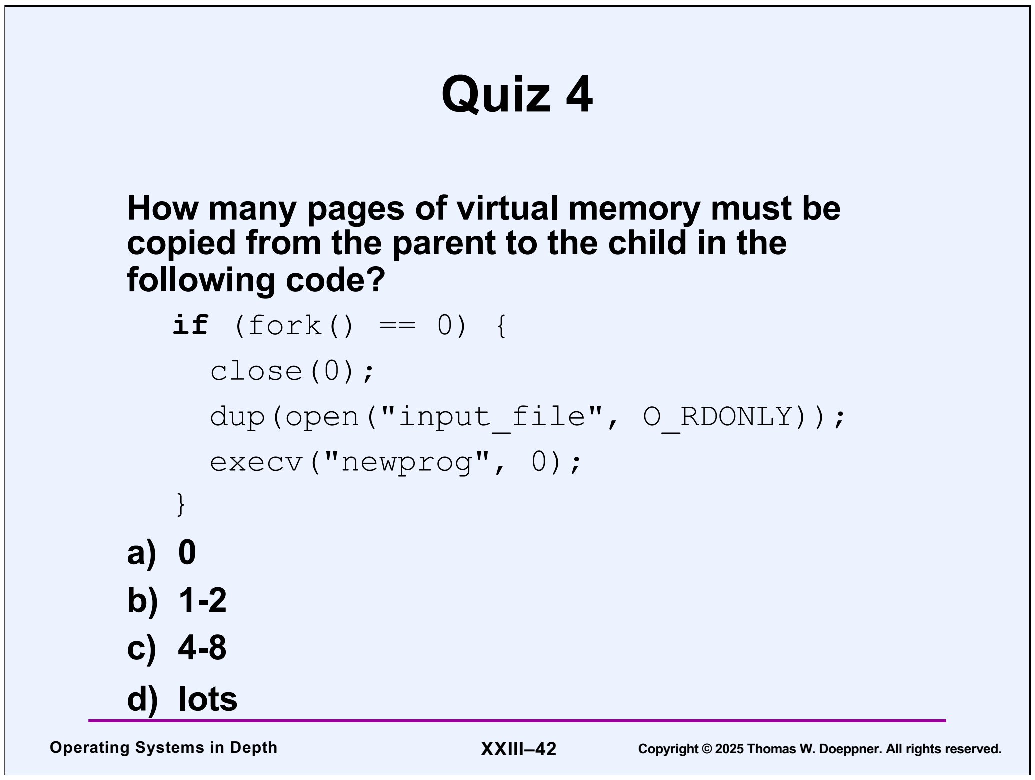
Page frames in the **active** state hold pages that are in either a process's working set or the system's working set. The balance-set manager takes page frames out of this state and puts them into either the **standby** or **modified** states, corresponding to their being on the standby or modified lists. These lists contain threads that have not be used recently, but have not yet been made free (they are allowed to exist for a short additional period just in case there is a late reference to them). Two **page-writer threads** take page frames in the **modified** state, write them to secondary storage, and then put them in the **standby** state. The zero-page thread moves page frames from the **free** state to the **zeroed** state. When the contents of a page frame are no longer useful the page frame is moved, by whatever thread discovers or causes the uselessness, to the **free** state. Pages are preferentially allocated from the free list. If that is empty, then they may be taken from the standby list.



Unix and Virtual Memory: The *fork/exec* Problem

- Naive implementation:
  - fork actually makes a copy of the parent's address space for the child
  - child executes a few instructions (setting up file descriptors, etc.)
  - child calls exec
  - result: a lot of time wasted copying the address space, though very little of the copy is actually used

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



Quiz 4

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

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

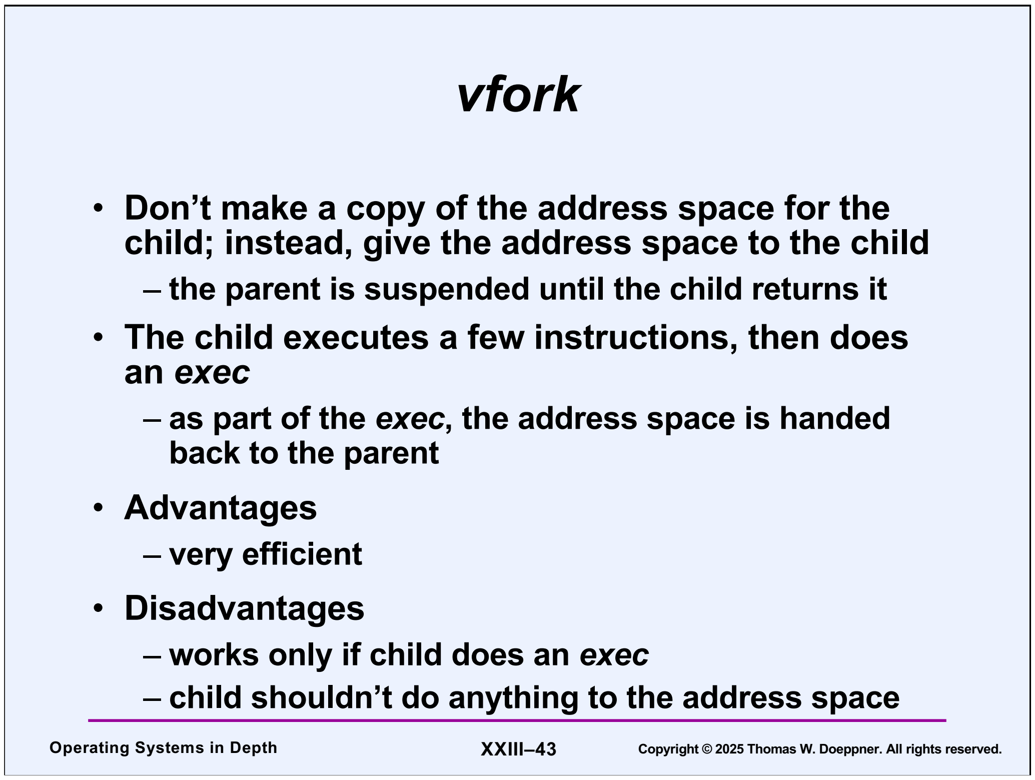
a) 0

b) 1-2

c) 4-8

d) lots

Assume the naïve implementation of fork. Also assume the parent process is at least of moderate size.



vfork

- Don't make a copy of the address space for the child; instead, give the address space to the child
  - the parent is suspended until the child returns it
- The child executes a few instructions, then does an *exec*
  - as part of the *exec*, the address space is handed back to the parent
- Advantages
  - very efficient
- Disadvantages
  - works only if child does an *exec*
  - child shouldn't do anything to the address space

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

Quiz 5

Will the assertion evaluate to true?

```
volatile int A = 6;
…
if (vfork() == 0) {
  A = 7;
  exit(0);
}
assert(A == 7);
…
```

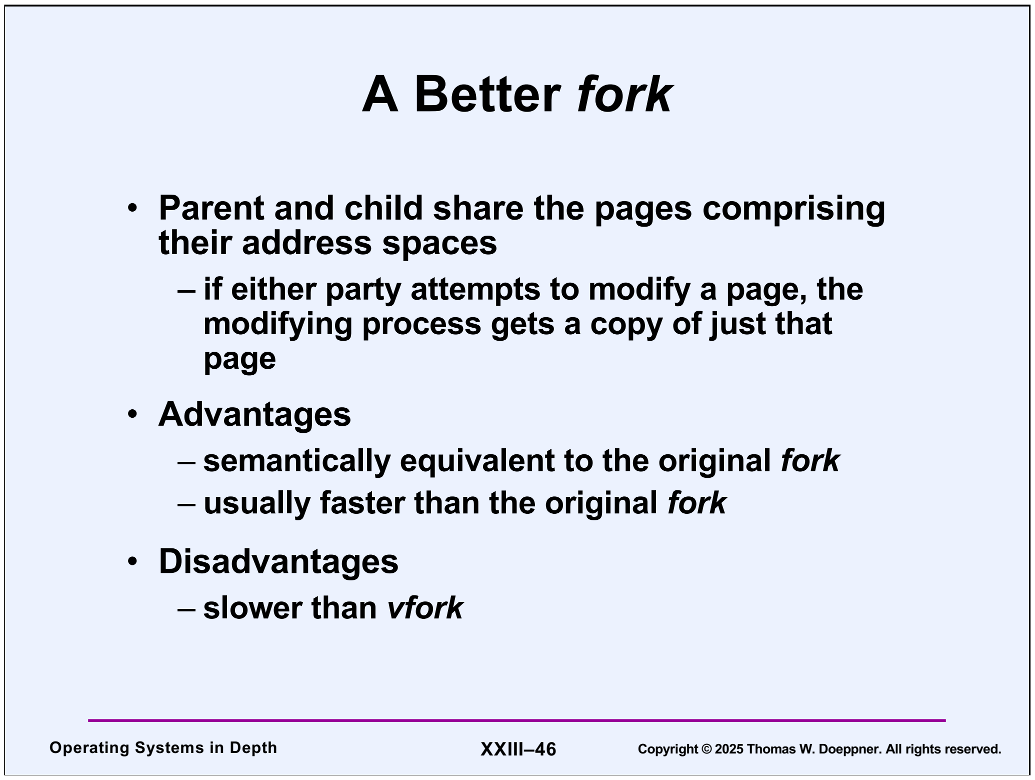
a) **it is never executed**

b) **it definitely won't evaluate to true**

c) **it will evaluate to true**

Lazy Evaluation

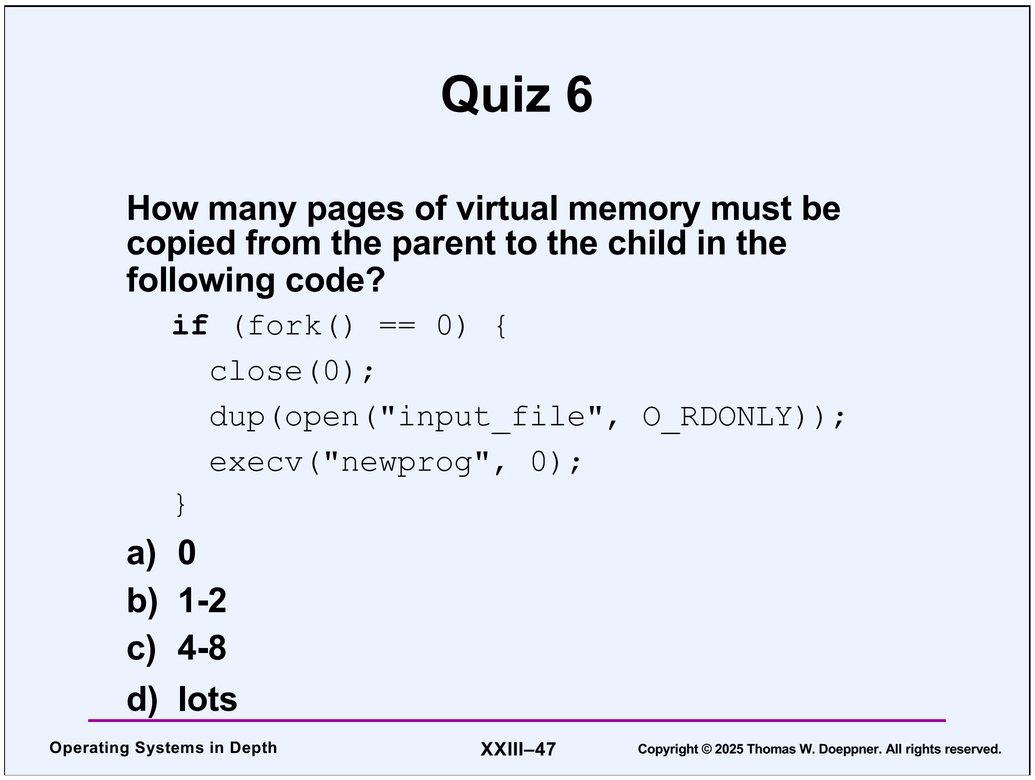
- Always put things off as long as possible
- If you wait long enough, you might not have to do them



A Better *fork*

- Parent and child share the pages comprising their address spaces
  - if either party attempts to modify a page, the modifying process gets a copy of just that page
- Advantages
  - semantically equivalent to the original *fork*
  - usually faster than the original *fork*
- Disadvantages
  - slower than *vfork*

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



Quiz 6

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

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

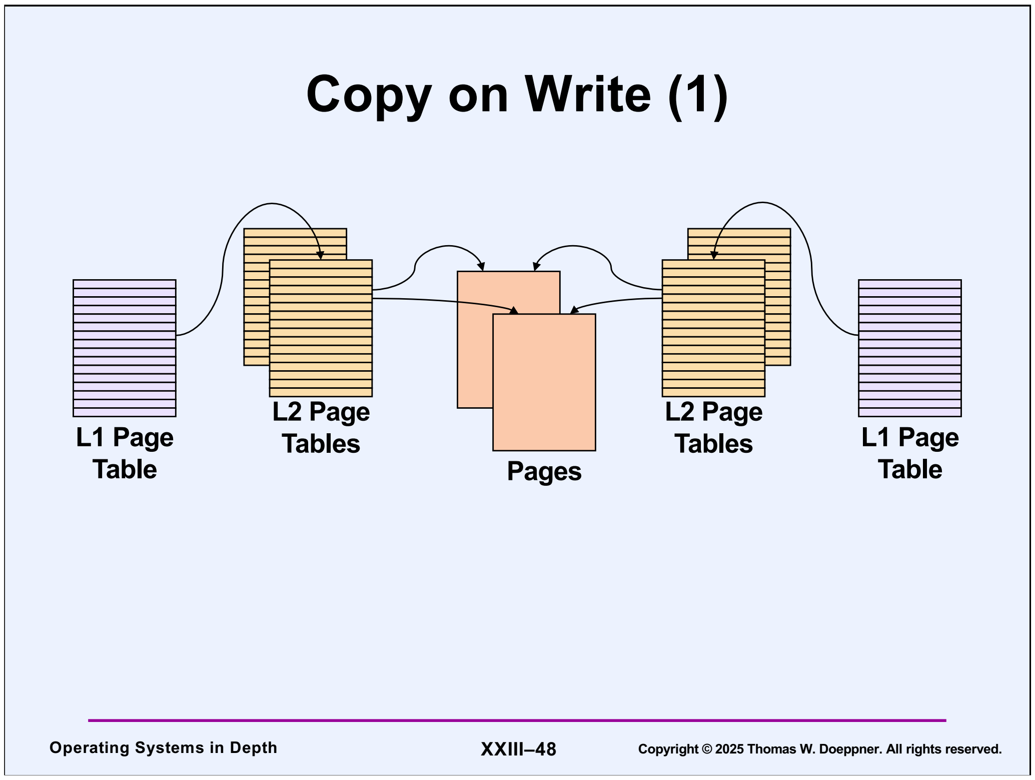
a) 0

b) 1-2

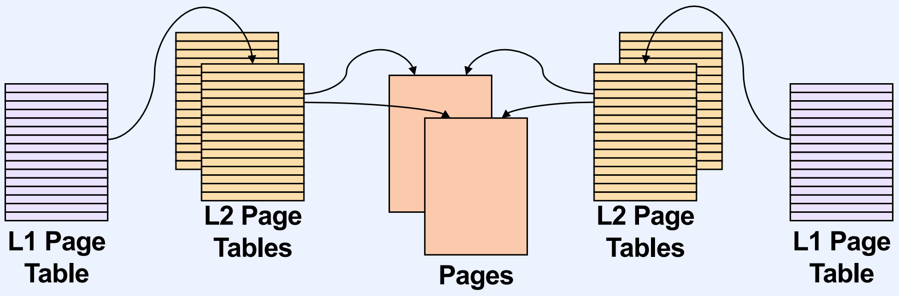
c) 4-8

d) lots

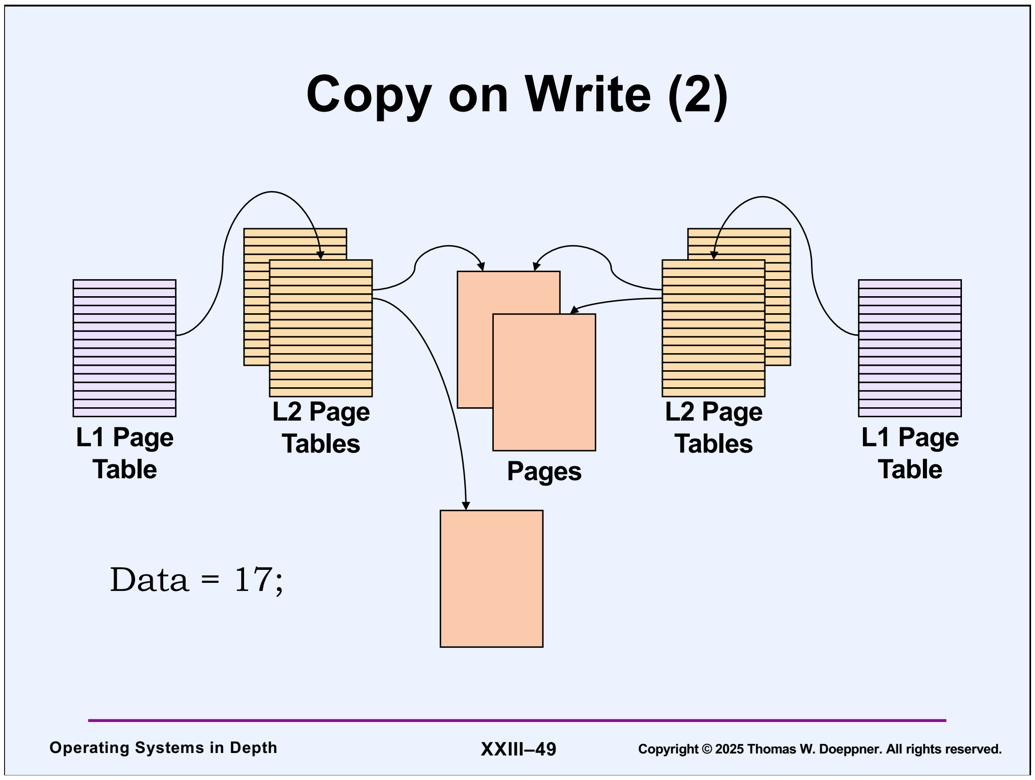
Assume a really good implementation of fork. Hint: keep in mind that we have to worry about text, data, and stack (among other possible regions).



Copy on Write (1)



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



Copy on Write (2)

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

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

Quiz 7

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

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

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

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