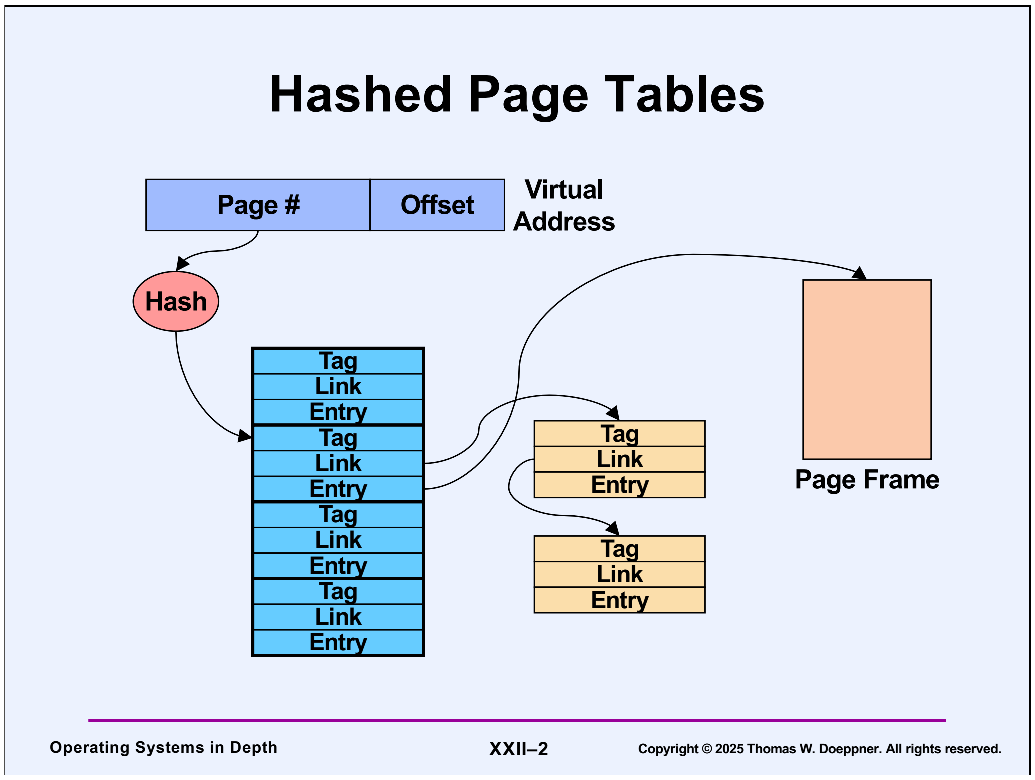
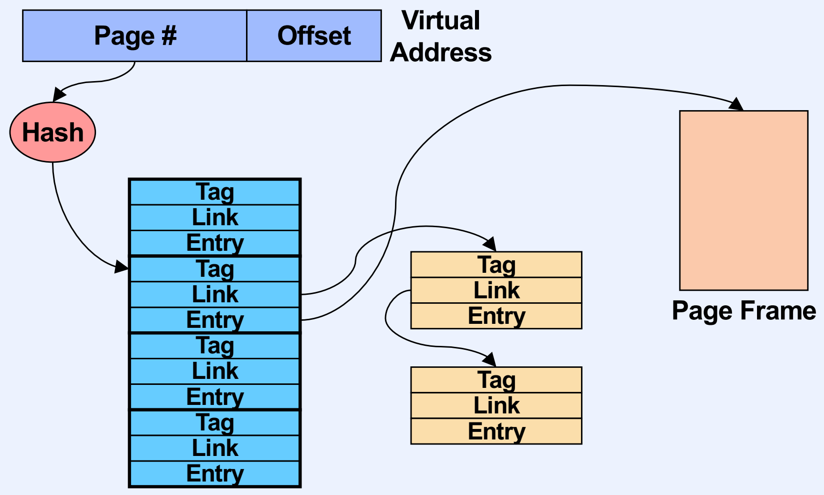
Memory Management Part 3

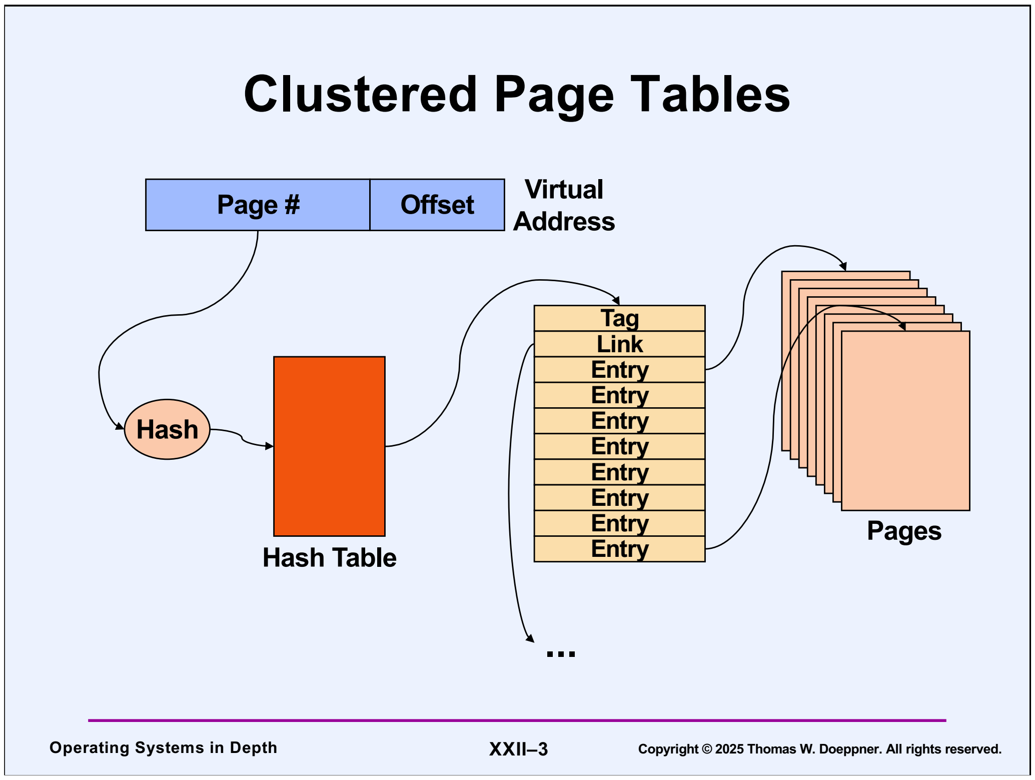


Hashed Page Tables



In a **hashed page table**, the page number is a key used as the entry into a hash table. Collisions are handled by chaining. In the form shown in the slide, each page requires three words to represent it.

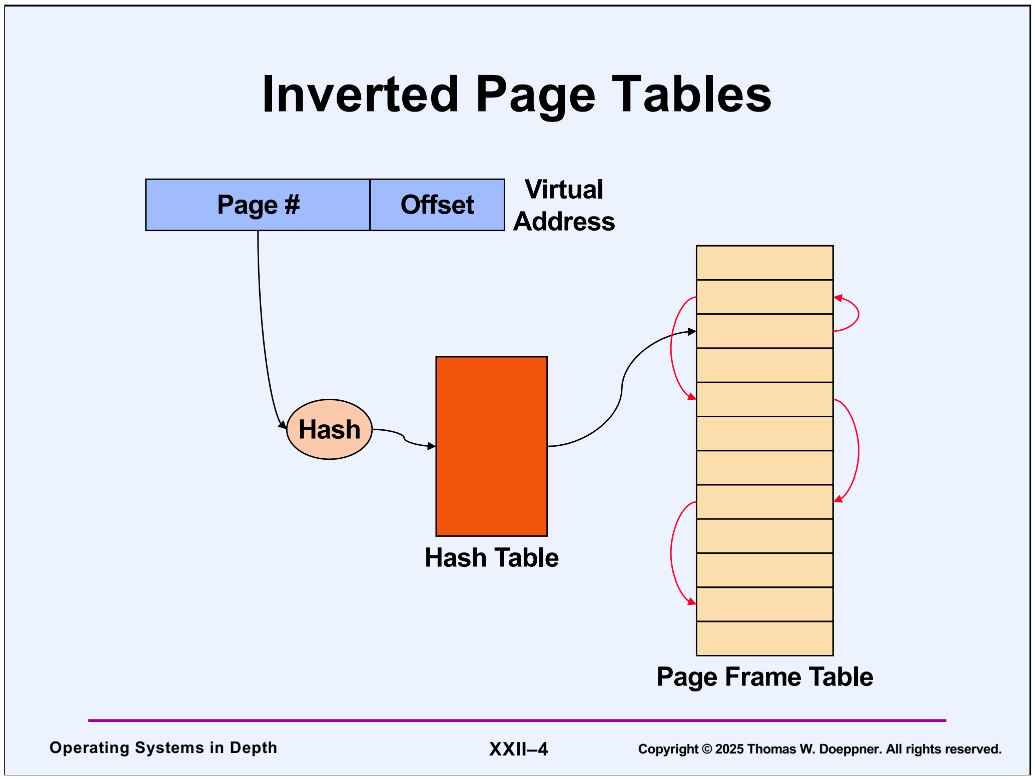
Hashed page tables support widely but sparsely allocated address spaces well. However, they may require multiple memory accesses for some translations (which can be minimized by using hardware TLBs). Furthermore, a fair amount of extra space is required for chaining the collisions.



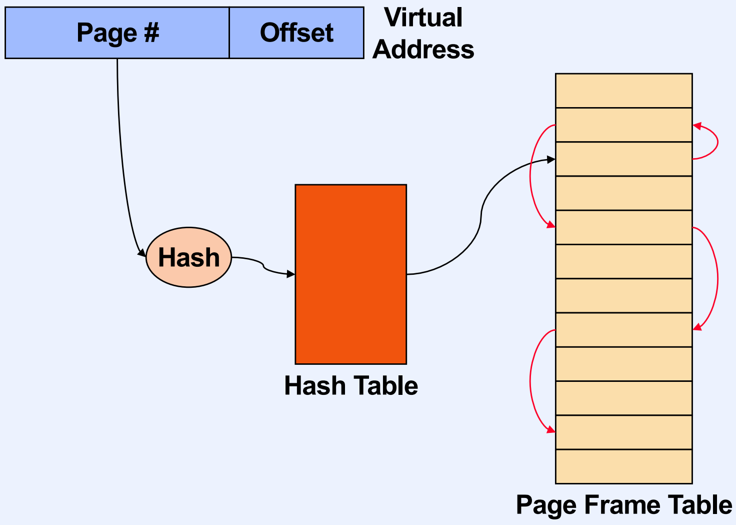
Clustered Page Tables

A variation of hashed paging that shows promise for supporting 64-bit architectures well is **clustered paging**. In this scheme, a number of pages (perhaps sixteen) are handled by each entry in the lists of hash synonyms. Thus there are three words of overhead per sixteen pages, rather than per page.

The paper "A New Page Table for 64-bit Address Spaces," by M. Talluri, M. Hill, and Y. Khalidi, **Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles**, December 1995, describes and analyzes this scheme.



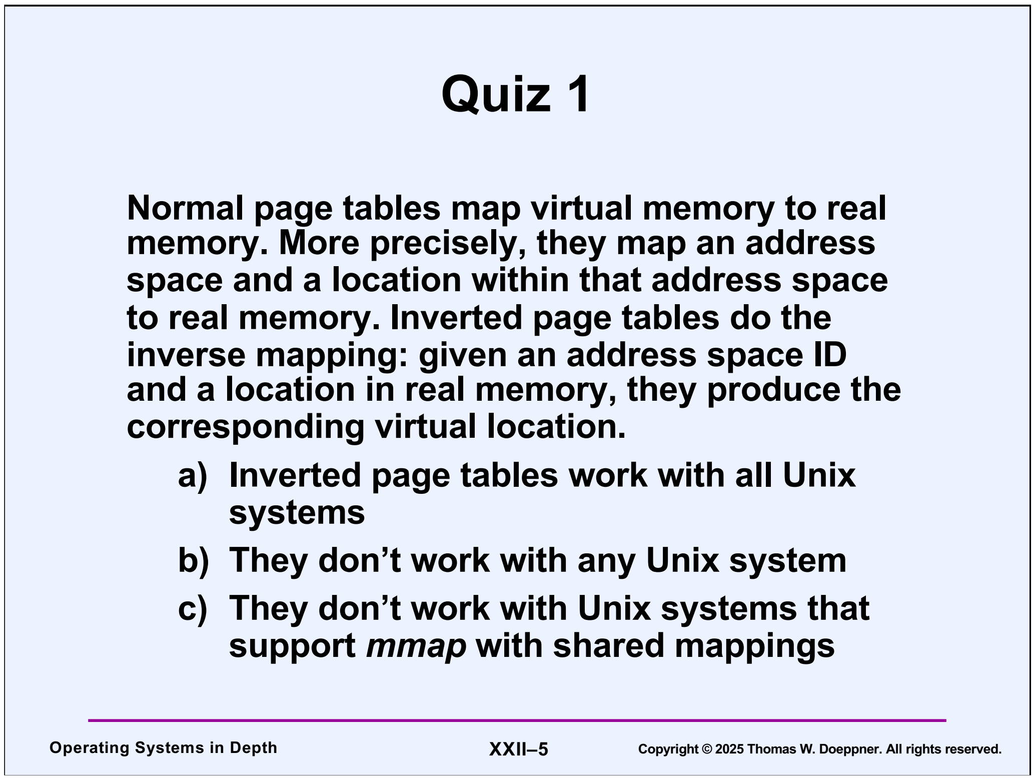
Inverted Page Tables



Here, **inverted page tables**, a variant of hashed page tables, are used to avoid wasting a large amount of memory for the translation map. A **page frame table** is maintained that indicates for each page frame of real memory what virtual address is mapped into it. In a typical implementation (we describe here a simplification of the IBM RS/6000 scheme), the hardware takes the page number from the virtual address, hashes this into a hash table, and then follows a linked list of hash synonyms in the page-frame table until it finds the desired entry. Then the index of this entry (in the page-frame table) is the page-frame number of the page. If the entry is not found, then a page fault is generated.

This procedure would be quite slow if it were always performed exactly as described. However, it can be combined with the use of a TLB to achieve a system that, on the average, performs well.

Another difficulty with inverted page tables is that there are usually portions of several address spaces in primary storage. Thus the virtual address of a page does not identify it uniquely, since different address spaces have pages with identical virtual addresses. So, there must be some sort of **address space ID** to indicate which page is whose. This is accomplished via a hardware register that contains the address space ID of the current address space, and each entry in the page-frame table also contains an address space ID.

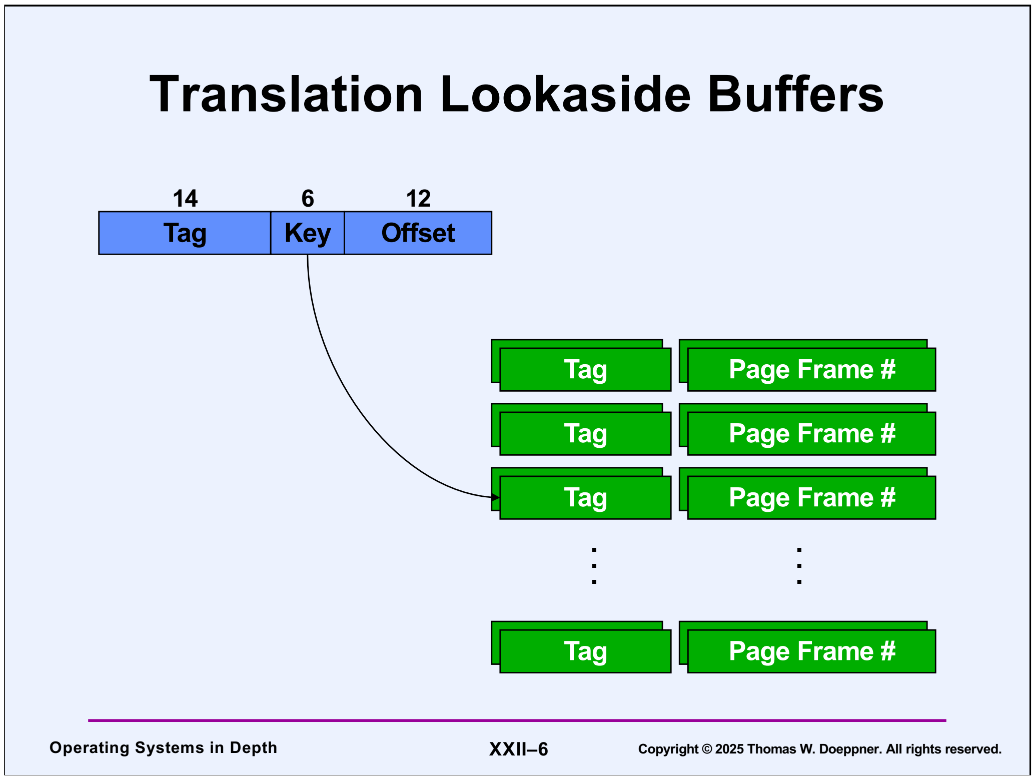


Quiz 1

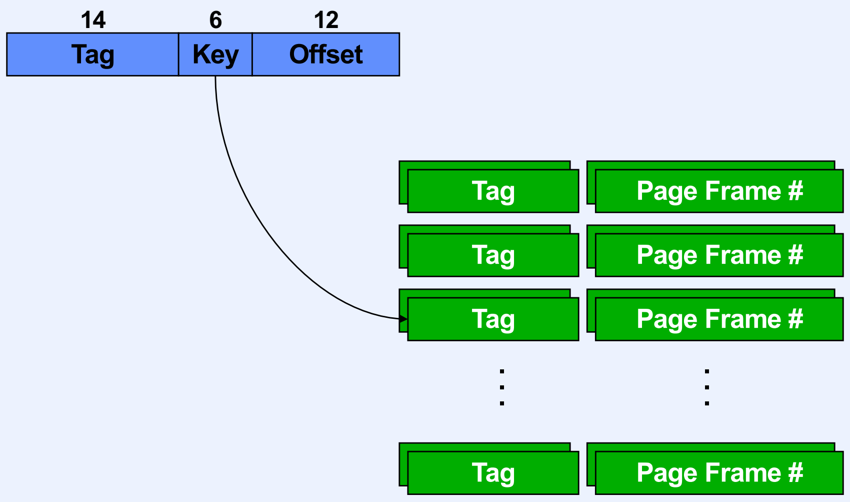
Normal page tables map virtual memory to real memory. More precisely, they map an address space and a location within that address space to real memory. Inverted page tables do the inverse mapping: given an address space ID and a location in real memory, they produce the corresponding virtual location.

a) Inverted page tables work with all Unix systems
b) They don’t work with any Unix system
c) They don’t work with Unix systems that support *mmap* with shared mappings

Hint: Do normal page tables (i.e. not inverted) implement a one-to-one function or a many-to-one function?



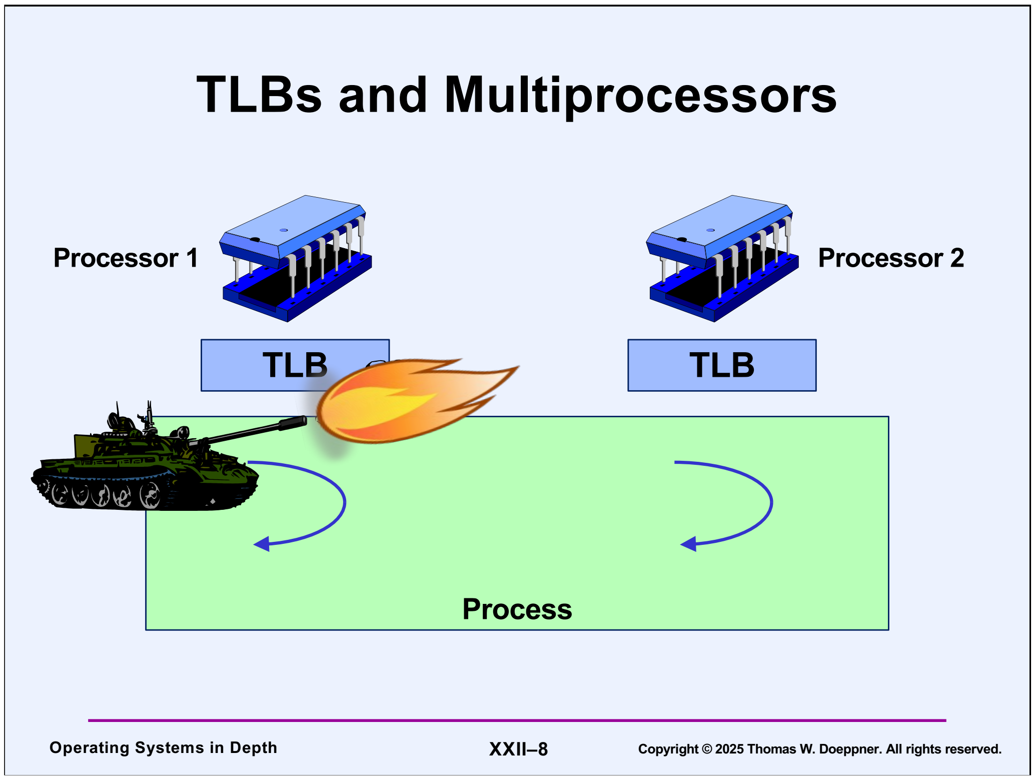
Translation Lookaside Buffers



Some architectures (e.g. MIPS) employ only TLBs for address translation. Thus, if there's a cache miss, the result is a page fault and the OS is called upon to find the page and insert its mapping into the cache.

TLBs and Mappings

- TLBs provide a cache for mappings from virtual addresses to real addresses
- Mappings change when
  - pages are removed (unmapped) from memory
  - when the address space is switched from one process's to another's
- OS must explicitly flush old contents of TLB
  - either individual entries or all of it



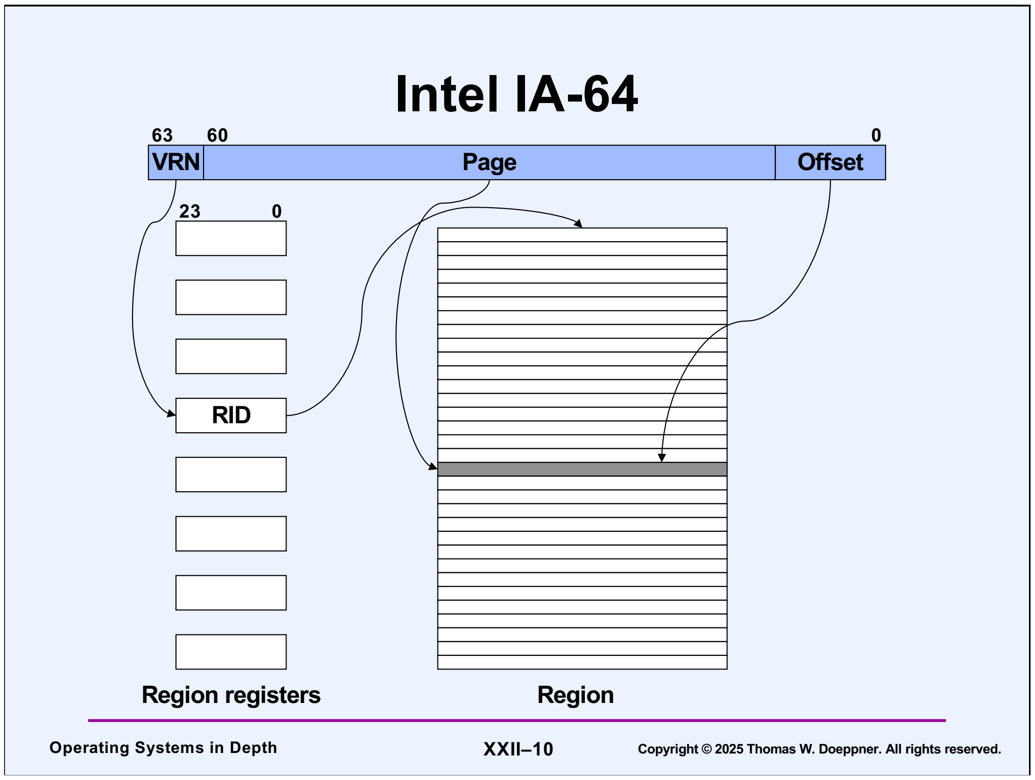
TLBs and Multiprocessors

For details, see the textbook, page 297.

TLB Shootdown Algorithm

```
// shooter code
for all processors i sharing
  address space
   interrupt(i);
for all processors i sharing
  address space
   while (noted[i] == 0)
      ;
modify_page_table();
update_or_flush_tlb();
done[me] = 1;
```

```
// shootee i interrupt handler
receive_interrupt_from_
  processor j
noted[i] = 1
while (done[j] == 0)
   ;
flush_tlb()
```



Intel IA-64

The detailed Intel documentation on the Intel IA-64 architecture was originally at https://www.intel.com/content/www/us/en/processors/itanium/itanium-architecture-vol-1-2-3-4-reference-set-manual.html. However, these links no longer. An article explaining the architecture can be found at https://courses.cs.washington.edu/courses/csep548/06au/readings/ia-64.pdf;

As the slide shows, a 64-bit virtual address is split into a 3-bit region number and a 61-bit region address. However, regions are assigned 24-bit IDs and thus there can be 2^{24} of them. The region-number field of the virtual address selects one of eight region registers, each of which contains a 24-bit ID of a region. Thus, the size of the address space can actually be 2^{85} bytes. The page size varies (though is fixed per region) from 4 kilobytes to 256 megabytes.

A better way of thinking about IA-64 addressing is that it supports 2^{24} separate 61-bit address spaces. The 61-bit address space is identified by the 24-bit region number. Some of these regions might be shared among multiple processes, others might be private. When the processor switches from one process to another, rather than flushing the entire TLB, the region numbers are used to identify which translations are for private regions, and thus should be flushed, and which translations should stay in the TLB, since they might be used by the process being switched into.

A lengthy discussion of how virtual memory was implemented on the IA-64 implementation of Linux can be found at https://www.informit.com/articles/article.aspx?p=29961.

Quiz 2

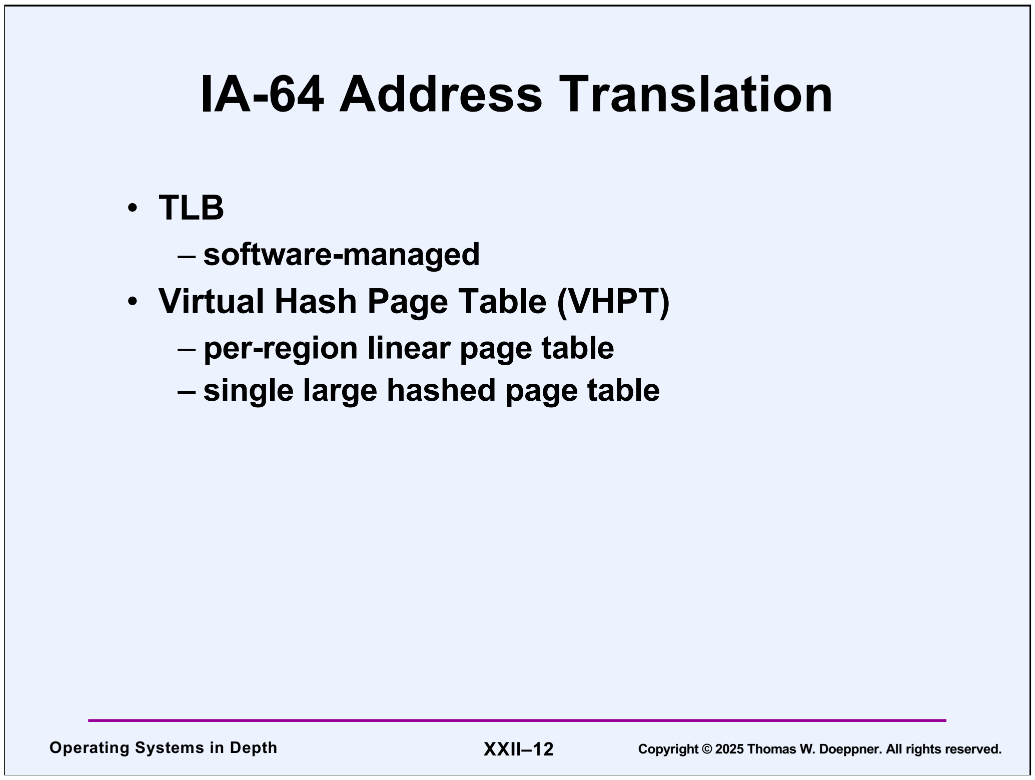
What did the IA-64 designers have in mind as a use for the region registers?

a) Region registers facilitate increasing the sizes of regions

b) They felt that 2^{64} bytes might be too small a limit for address-space size; the region registers provide a means for making it larger

c) Region registers are a means for identifying and mapping in shared objects, such as libraries or datasets

d) Region-register contents are used in the TLB to uniquely identify regions; TLB flushes are minimized on process switches because it’s clear which process an address belongs to



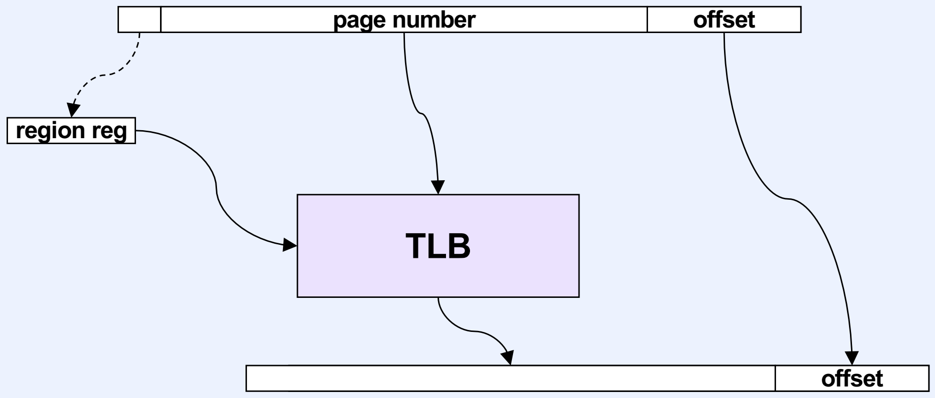
IA-64 Address Translation

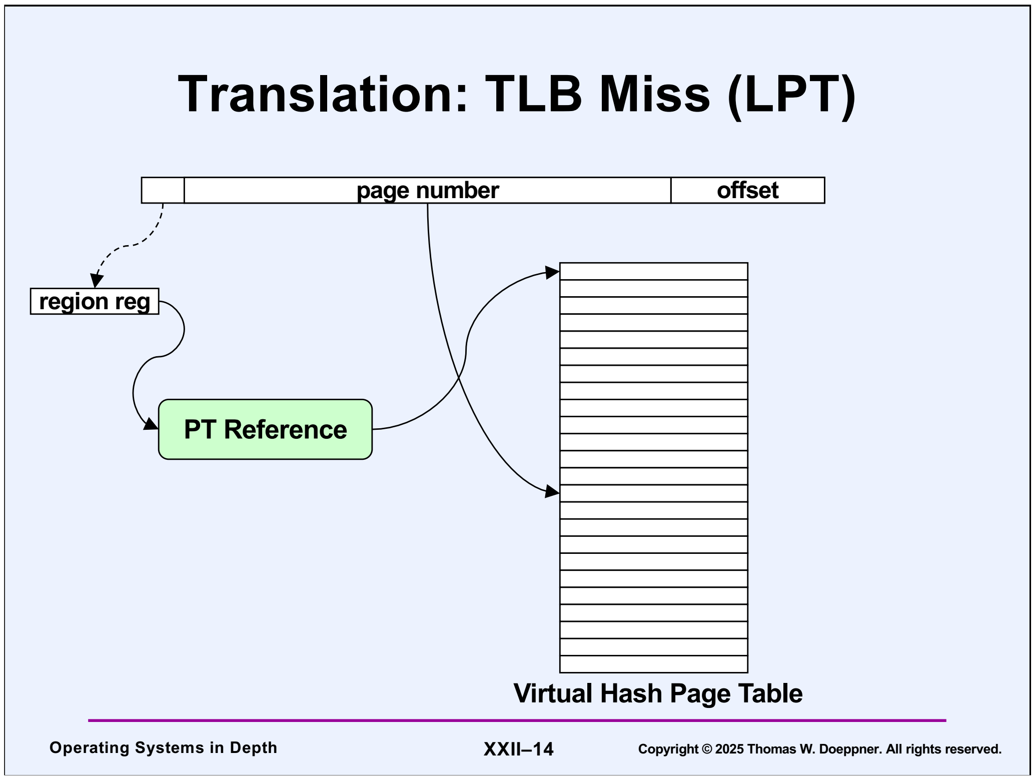
- TLB
  - software-managed
- Virtual Hash Page Table (VHPT)
  - per-region linear page table
  - single large hashed page table

The IA-64 architecture provides a software-managed TLB to handle translation. On TLB misses, an OS can arrange for the hardware to use a virtual hash page table (VHPT) to lookup the translation and reload the cache. There are two options for the VHPT: it can be a per-region linear page table or a single large hashed page table. If the former, then each region has its own linear page table within the region address space. On TLB misses, the hardware looks up the address within the appropriate linear page table. Since the page table itself is in virtual memory, the addresses of its entries are translated by the TLB, which could result in another miss (which, in this case, would be handled directly by the OS).

If a single large hashed page table is used, then an implementation-defined hash function is used to map a virtual address into the (single) hash table, which is itself in virtual memory (and, again, the addresses of its entries are in virtual memory, thus possibly resulting in another TLB miss).

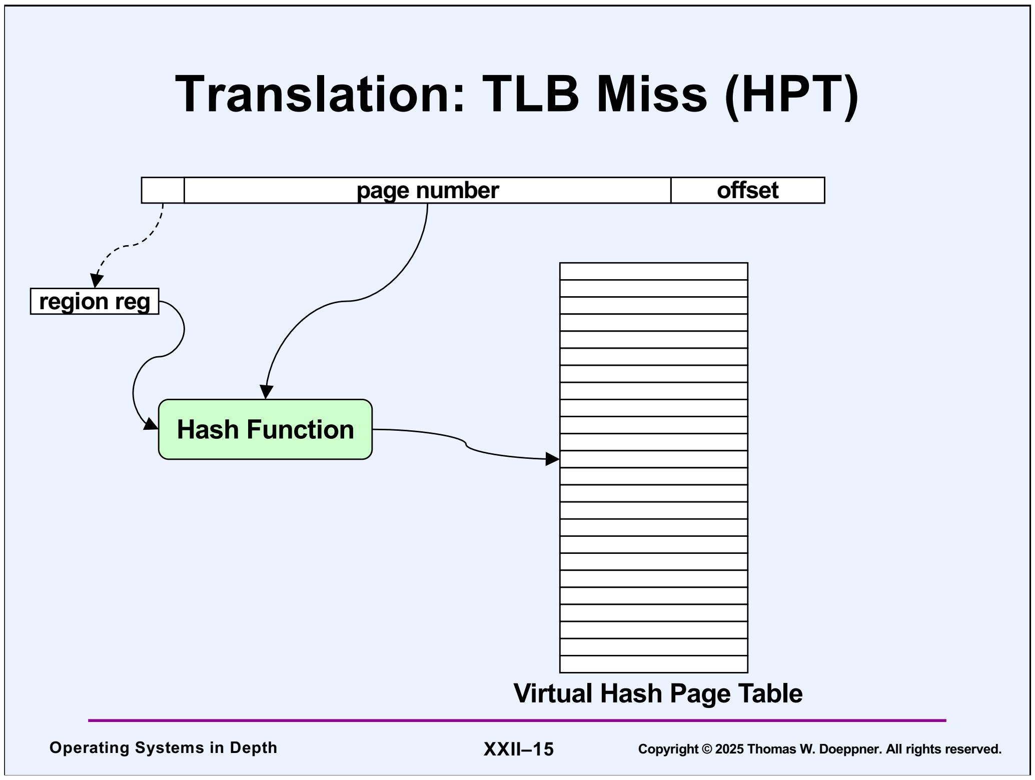
Translation: TLB



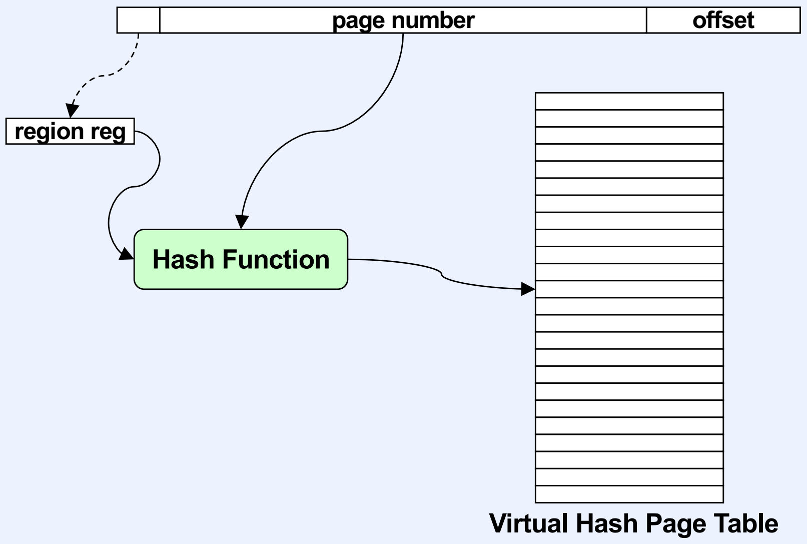


Translation: TLB Miss (LPT)

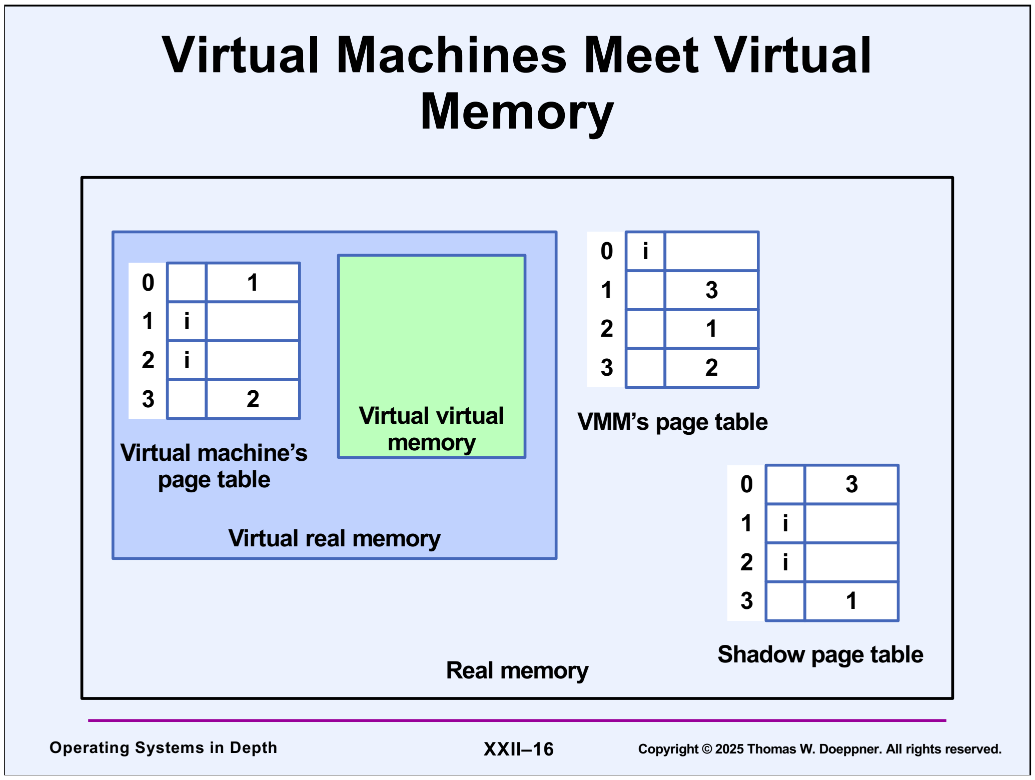
This slides shows how page translations are done using a per-region linear page table.



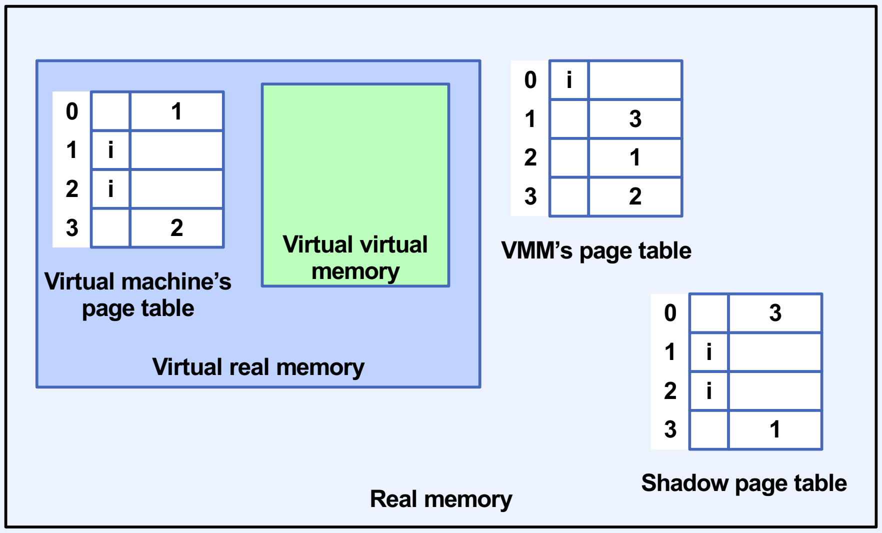
Translation: TLB Miss (HPT)



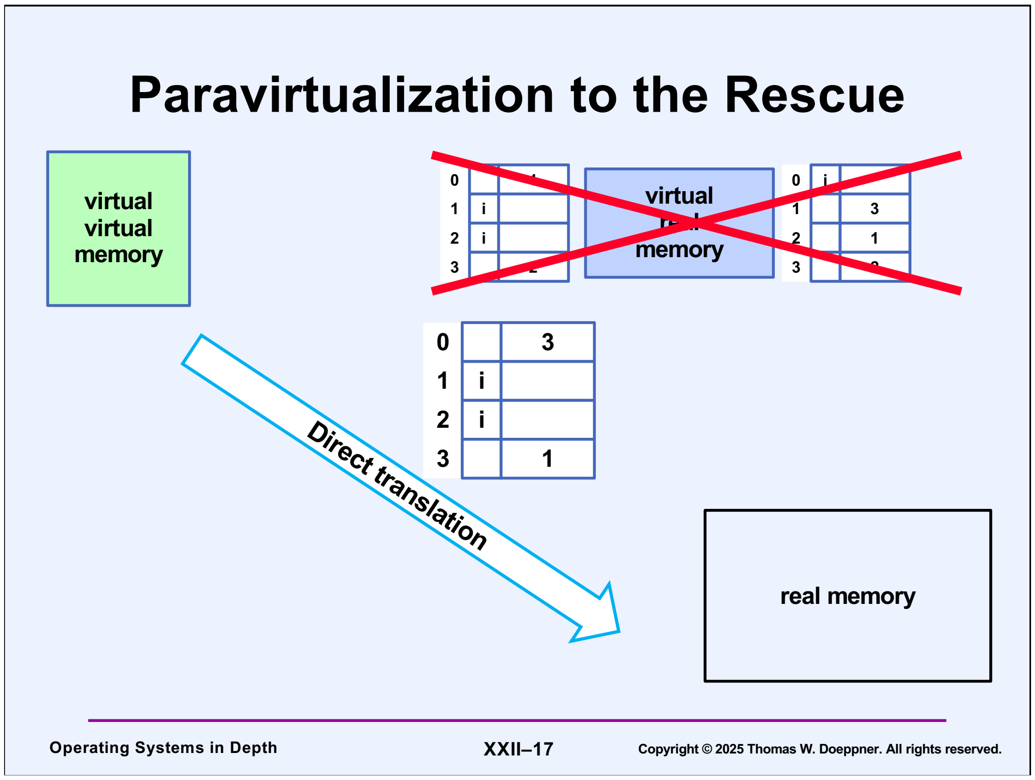
This slide shows how page translation is done via a single hashed page table. In this case, the hashed page table supplies address translation for the 85-bit address space.



Virtual Machines Meet Virtual Memory



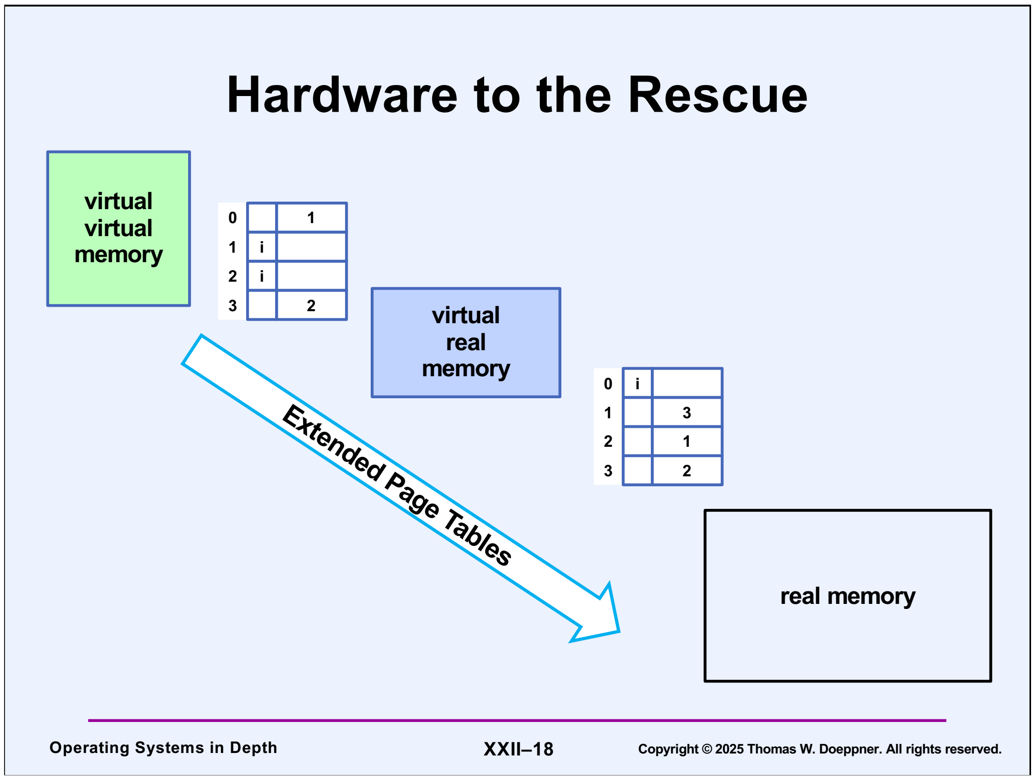
See the text Section 7.2.6, starting on page 299.



Paravirtualization to the Rescue

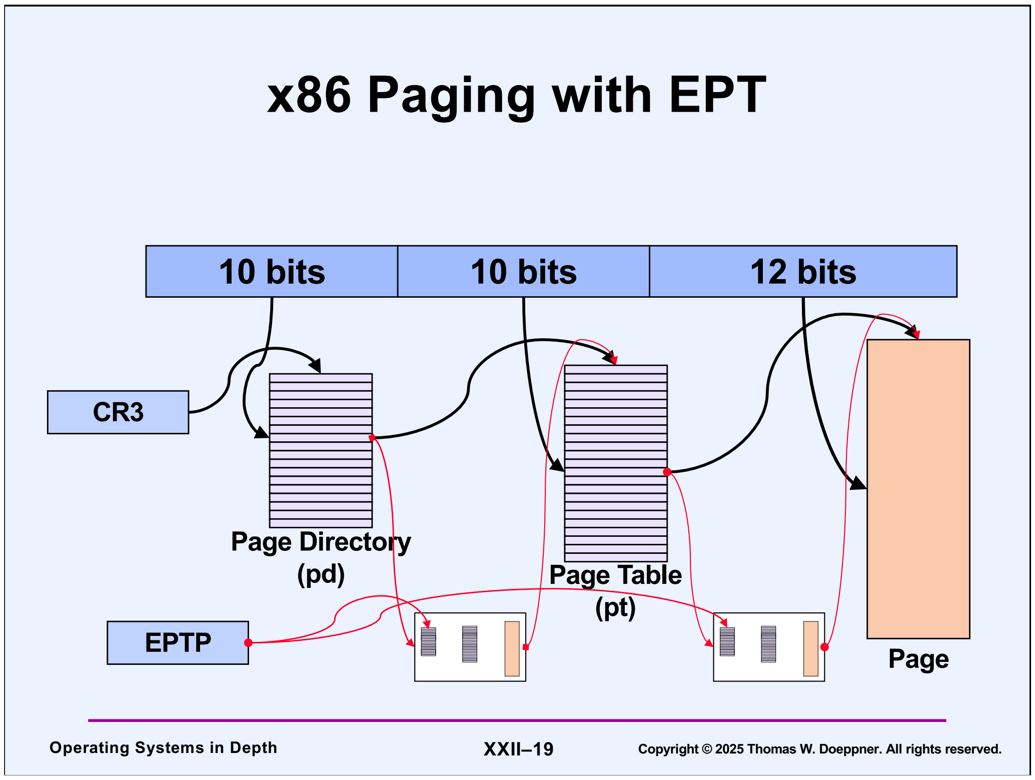
0		3
1	i	
2	i	
3		1

With paravirtualization (as in Xen), the guest OS in the virtual machine can cooperate with the VMM to produce a direct translation from virtual virtual memory to real memory.



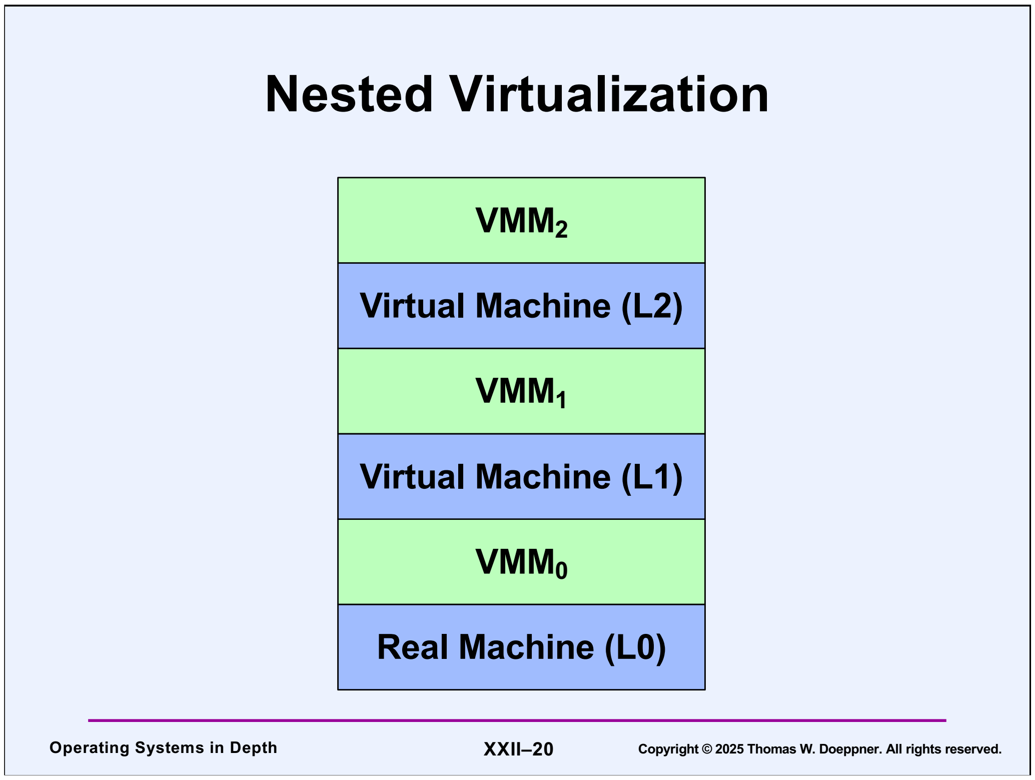
Hardware to the Rescue

Recent versions of the x86 architecture from Intel provide extended page tables (EPT) that add an extra layer of translation that's invisible to the virtual machine. The guest OS in the virtual machine sets up the normal hardware page tables to translate from what it considers to be virtual memory (actually virtual virtual memory) to what it considers to be real memory (actually virtual real memory). The VMM sets up an extra translation step from virtual real to real. The hardware does the composite translation (i.e., it uses both page tables).

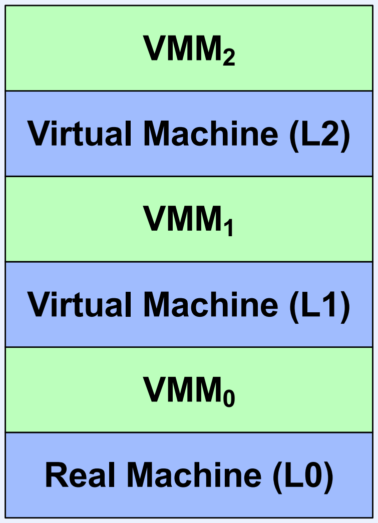


x86 Paging with EPT

On the actual x86 architecture, the translation (by EPT) from virtual real to real must be done at each step in which a real address is needed. The VMM maintains the EPT base pointer register (EPTP), which points to a page directory and its associated page tables that map virtual-real memory to real memory. When the guest OS puts the (virtual real) address of what it wants to be the page directory into CR3, the address is automatically translated by EPT into a real address. When a full translation is done (from virtual virtual to real real), the entry in the page directory containing the virtual real address of the page table is translated by EPT to a real address, and similarly for the entry in the page table.



Nested Virtualization



This material is based on “The Turtles Project: Design and Implementation of Nested Virtualization”, which can be found at https://www.usenix.org/event/osdi10/tech/full_papers/Ben-Yehuda.pdf.

Quiz 3

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

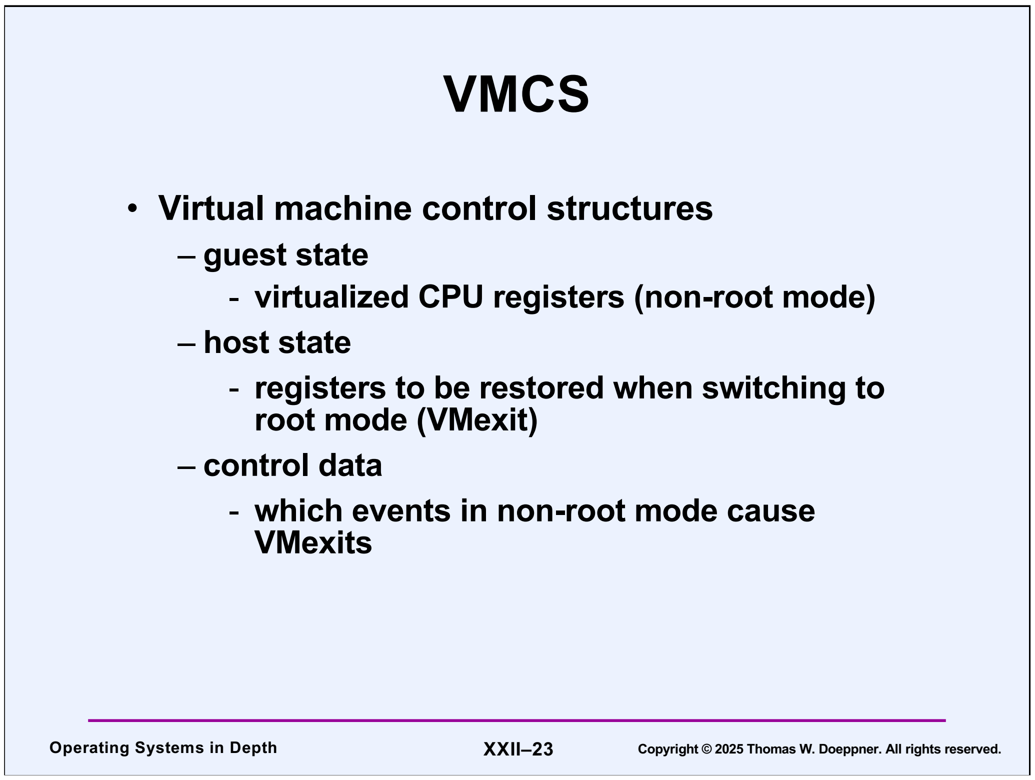
b) nothing: the EPT mechanism is virtualized by the hardware

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

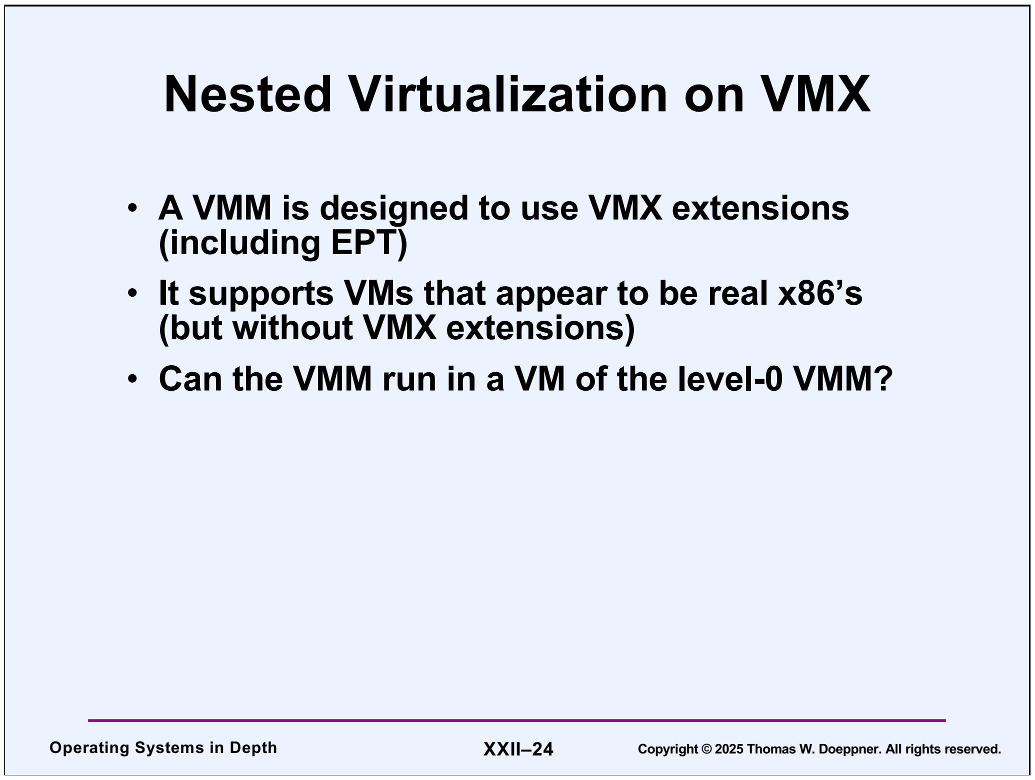
This is a review of what we've discussed before.



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

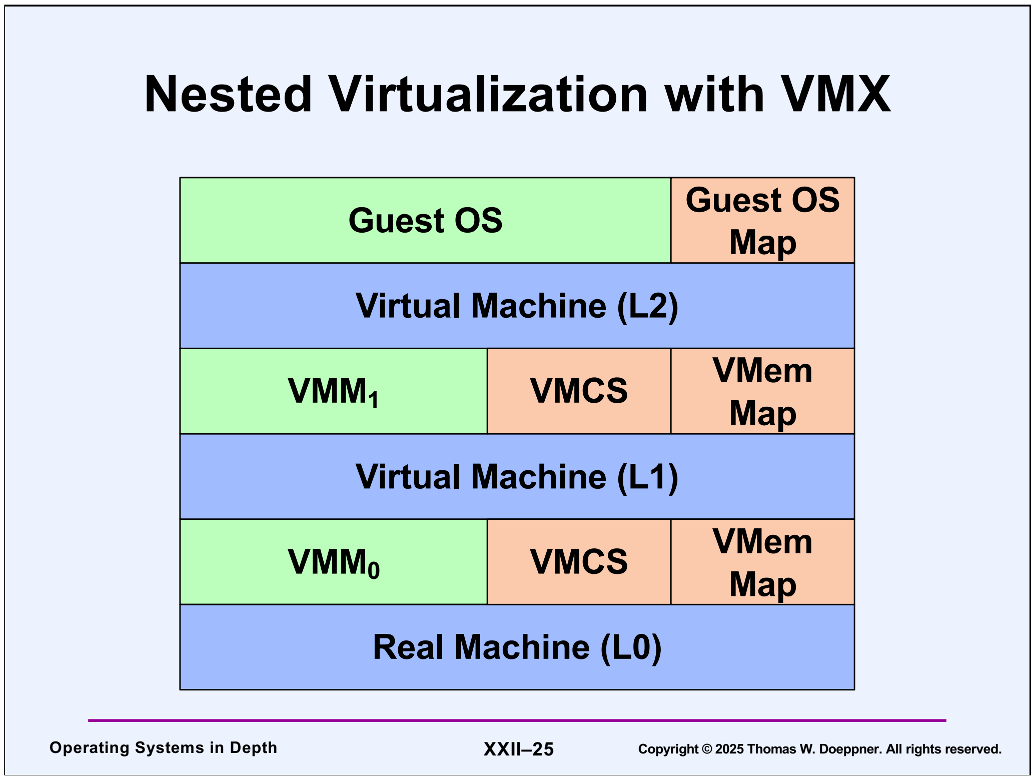
So is this.



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?

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.

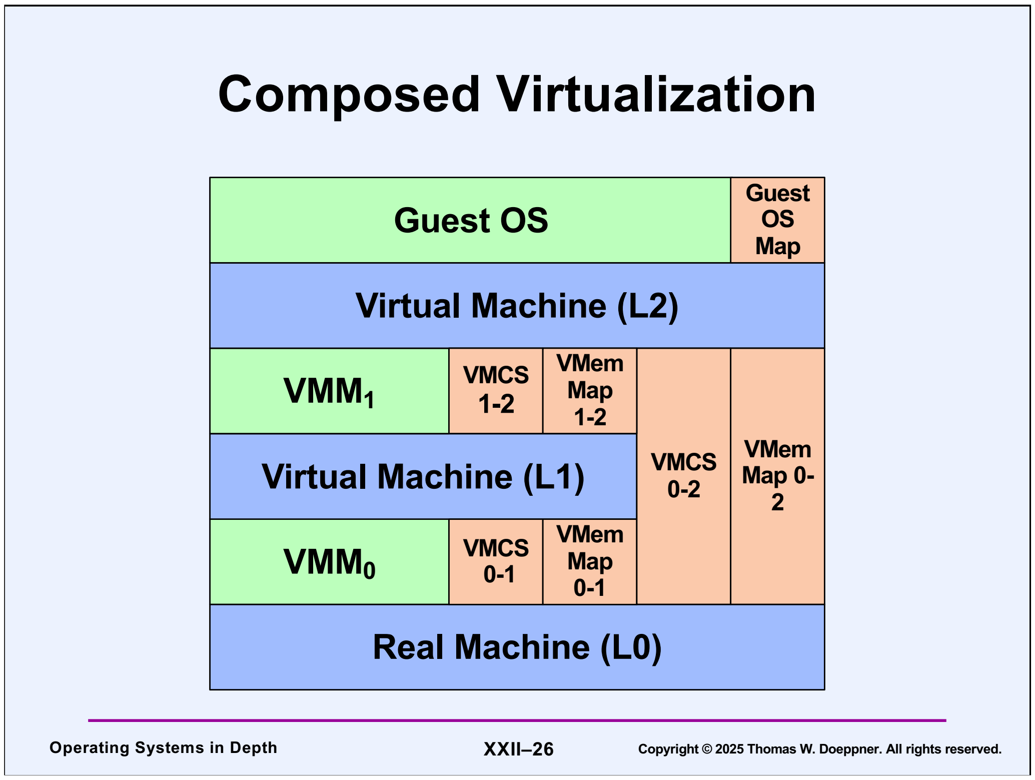


Nested Virtualization with VMX

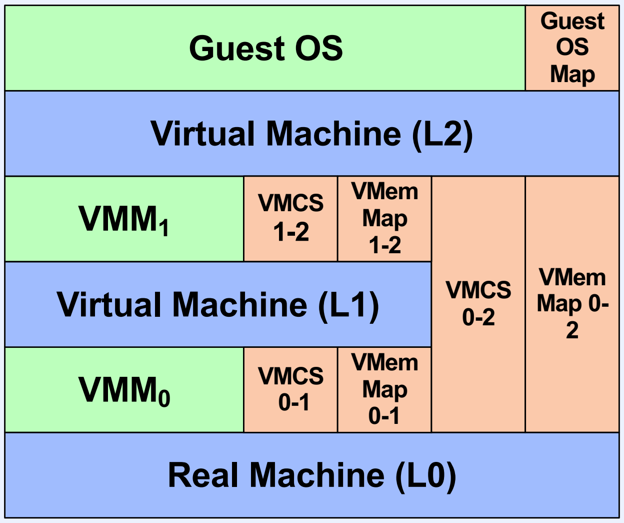
The slide illustrates what would happen in a "straightforward" approach to doing nested virtualization by virtualizing the VMX extensions. We have a guest OS running in a virtual machine supported by a VMM at level 2, which in turn is running in a virtual machine supported by a VMM at level 1, which 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.



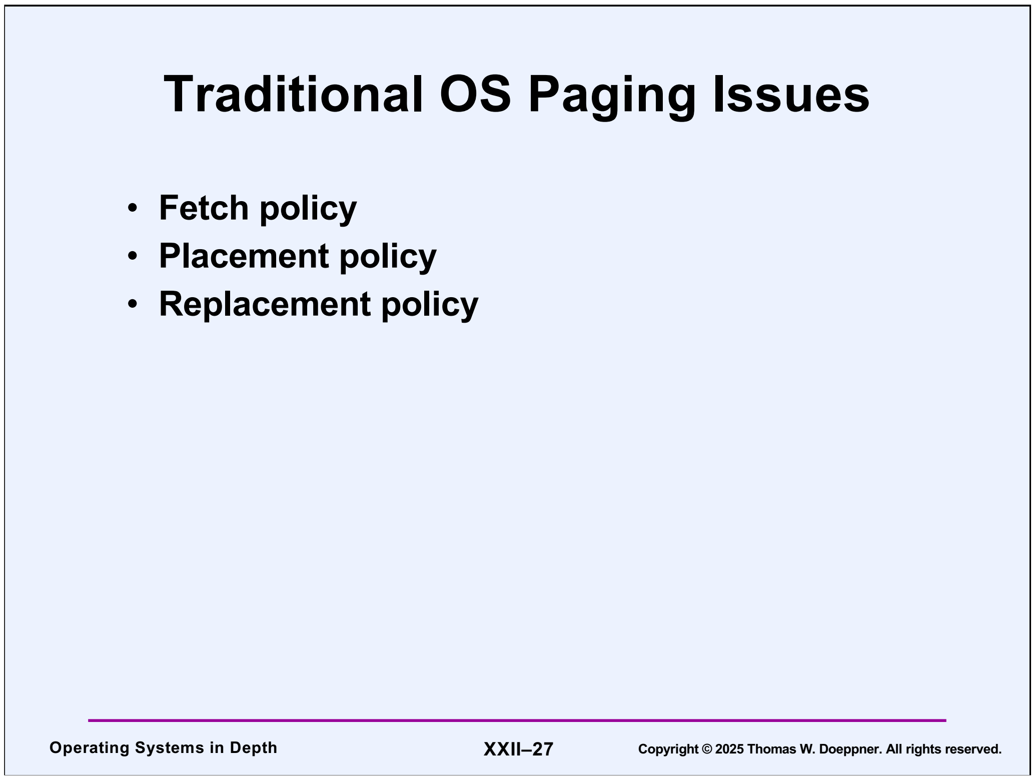
Composed Virtualization



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.

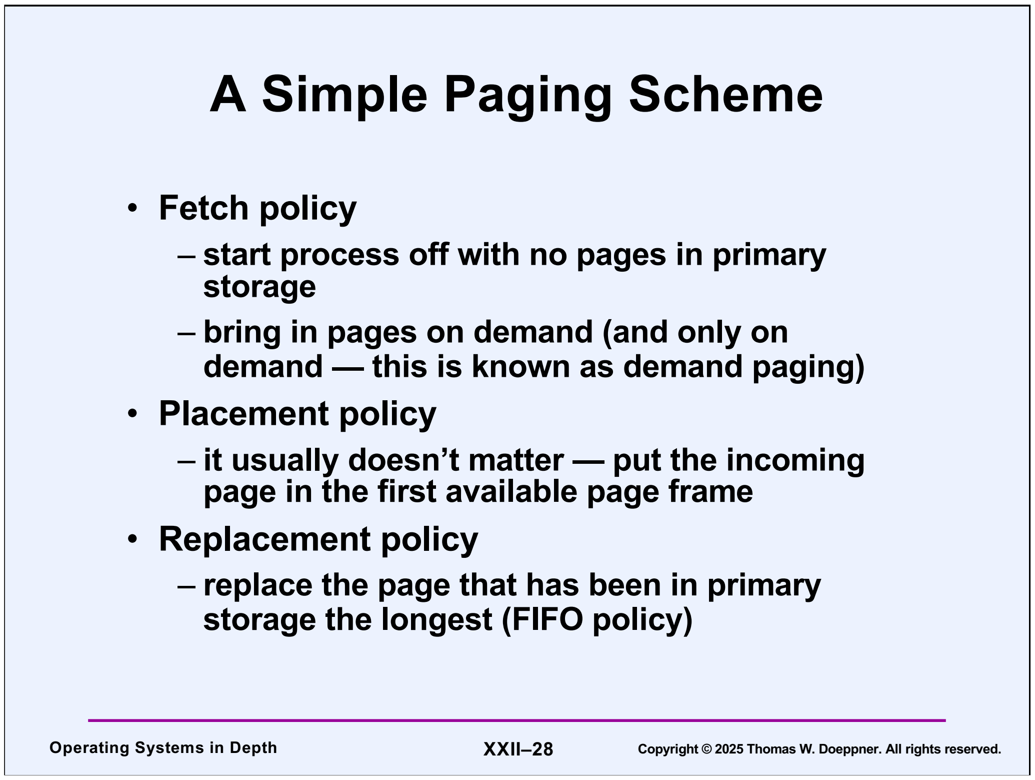
In the example shown in the slide, VMM$_1$ maintains VMCS$_{1\text{-}2}$ to represent the L2 virtual machine, while VMM$_0$ maintains both VMCS$_{0\text{-}1}$ to represent the L1 virtual machine and VMCS$_{0\text{-}2}$ to represent the L2 virtual machine. When VMM$_1$ attempts to do a VMenter with VMCS$_{1\text{-}2}$, a (real) VMexit occurs and control goes to VMM$_0$, which, in response, does a VMenter with VMCS$_{0\text{-}2}$. 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).



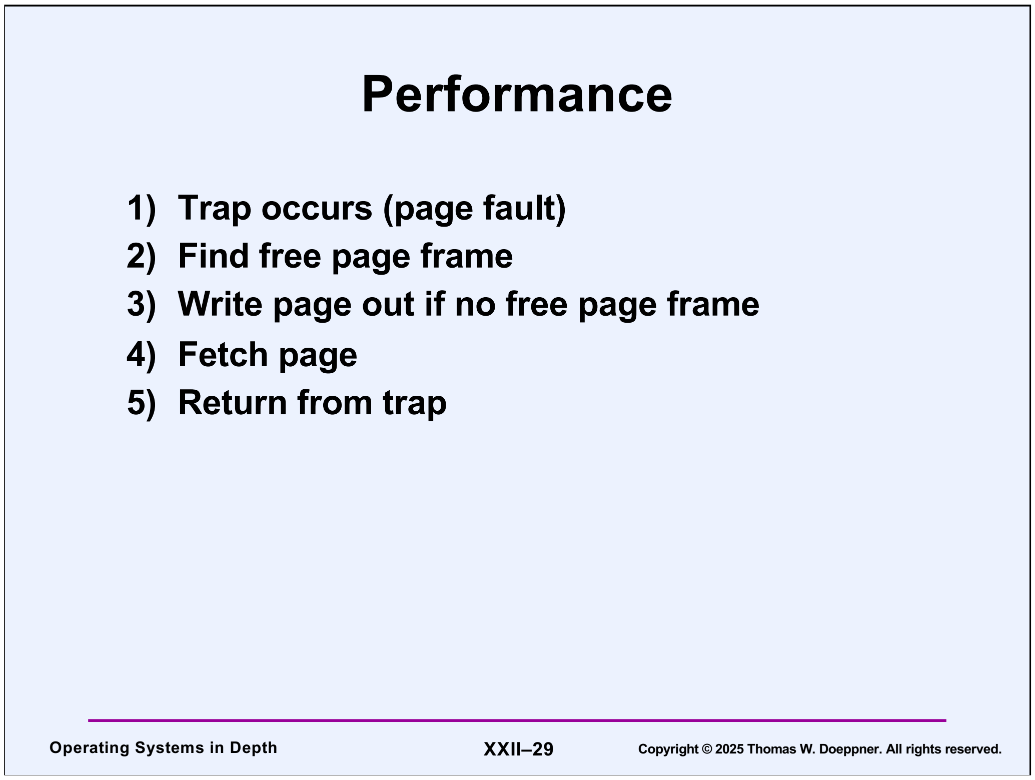
A Simple Paging Scheme

- 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.

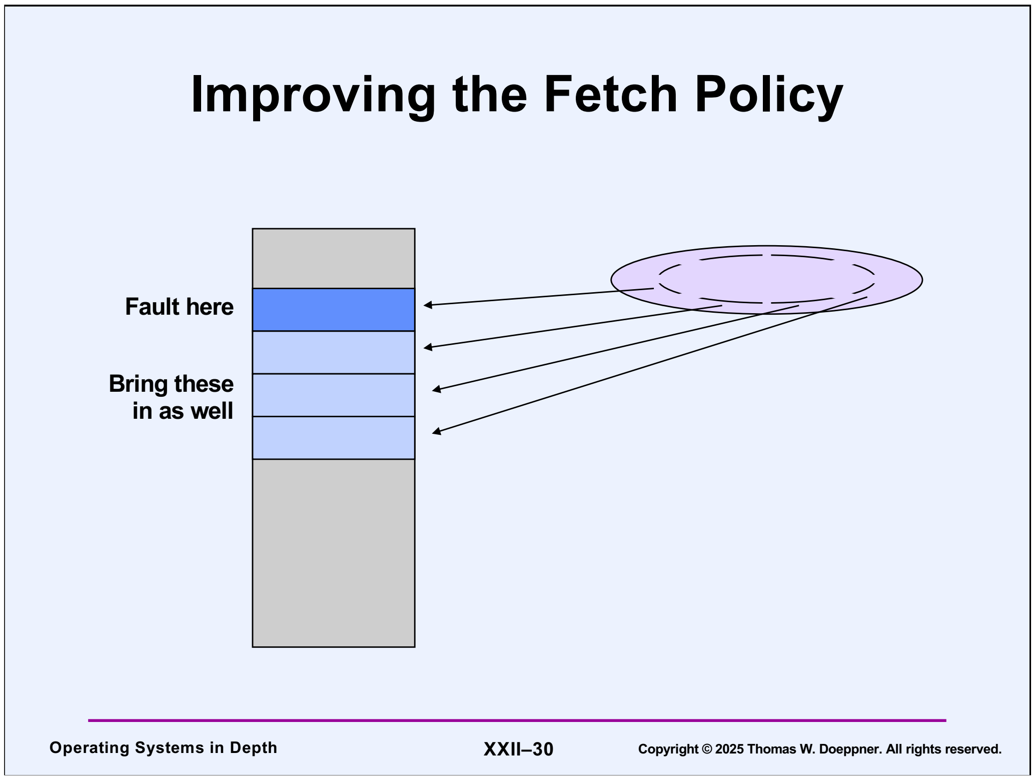
We start with a much too simple-minded replacement policy: when we want to fetch a page, but there is no room for it, we replace the page that has been in primary memory the longest. (This is known as a FIFO or first-in-first-out approach.)



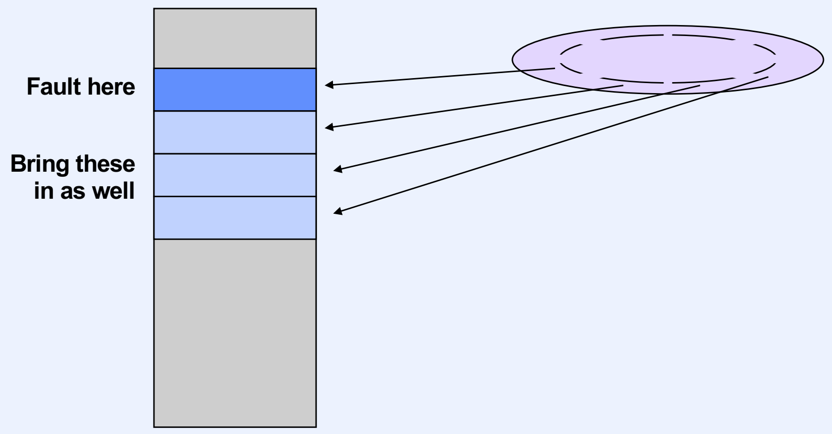
Performance

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.



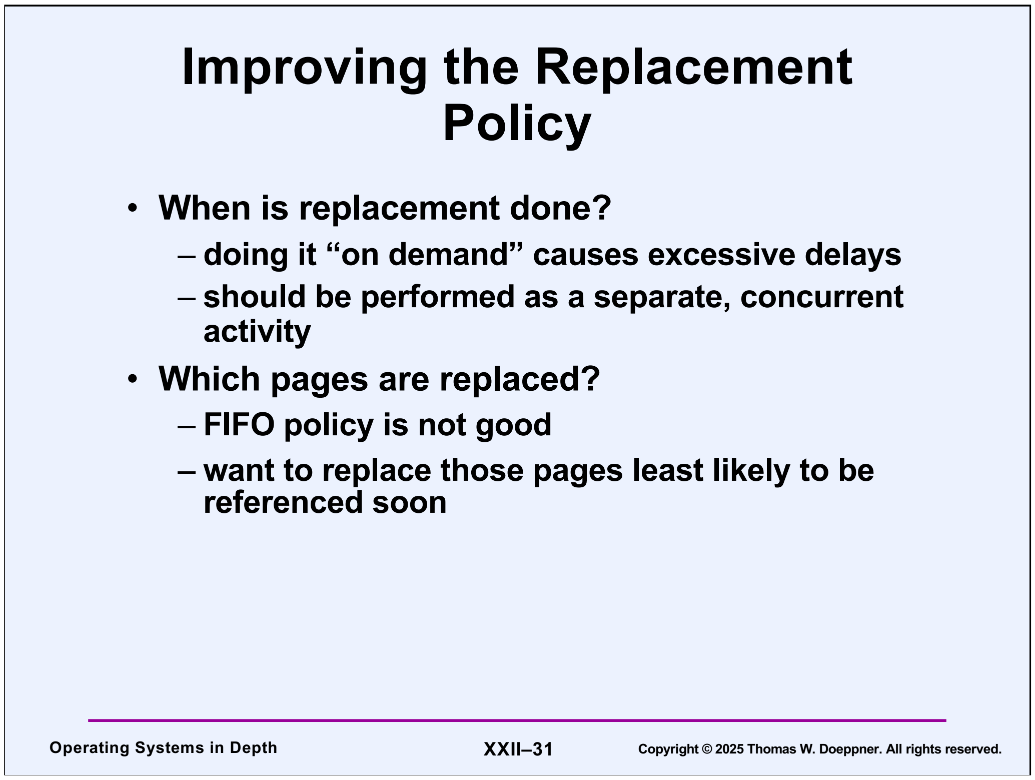
Improving the Fetch Policy



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.



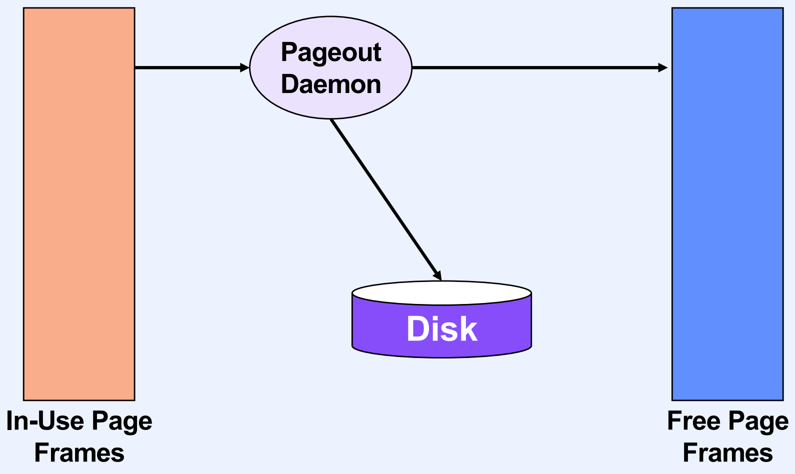
Improving the Replacement Policy

- 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.

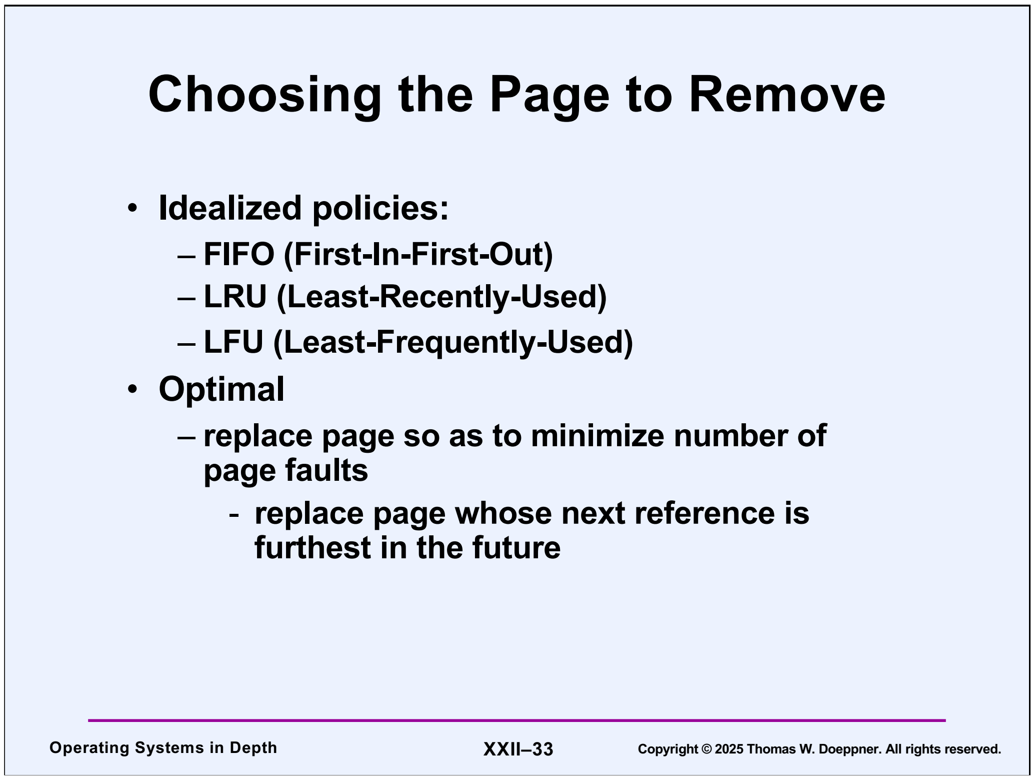
The other problem with the replacement policy is the use of the FIFO technique for deciding which pages to remove from primary storage—the page that has been in memory the longest could well be the page that is getting the vast majority of the references. Ideally, we would like to remove from primary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

Operating Systems in Depth XXII–33

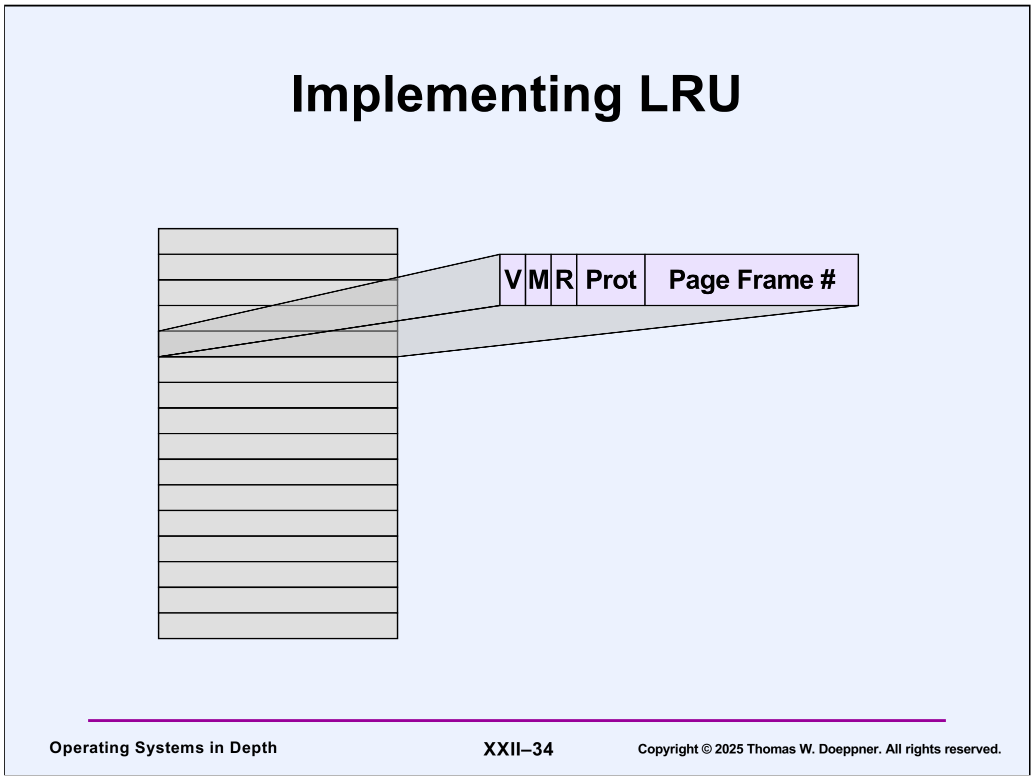
Determining which pages to replace is a decision that could have significant ramifications to system performance. If the wrong pages are replaced, i.e. pages which are referenced again fairly soon, then not only must these pages be fetched again, but a new set of pages must be tagged for removal.

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.

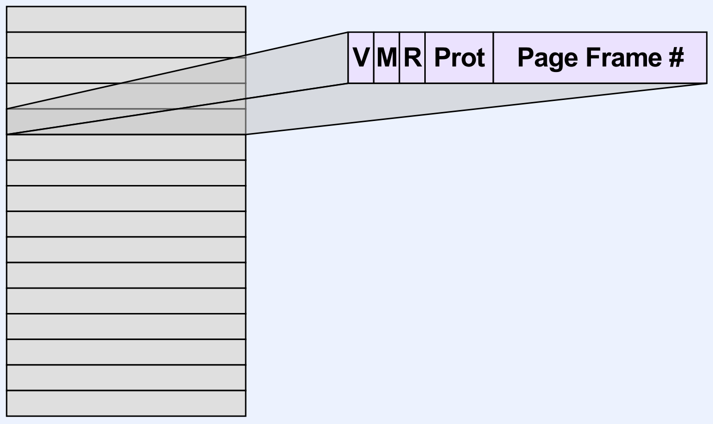
A variant of this approach is LFU—least frequently used. Pages are ordered by their frequency of use, and we replace the page which has been used the least frequently of those still in primary storage.

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.

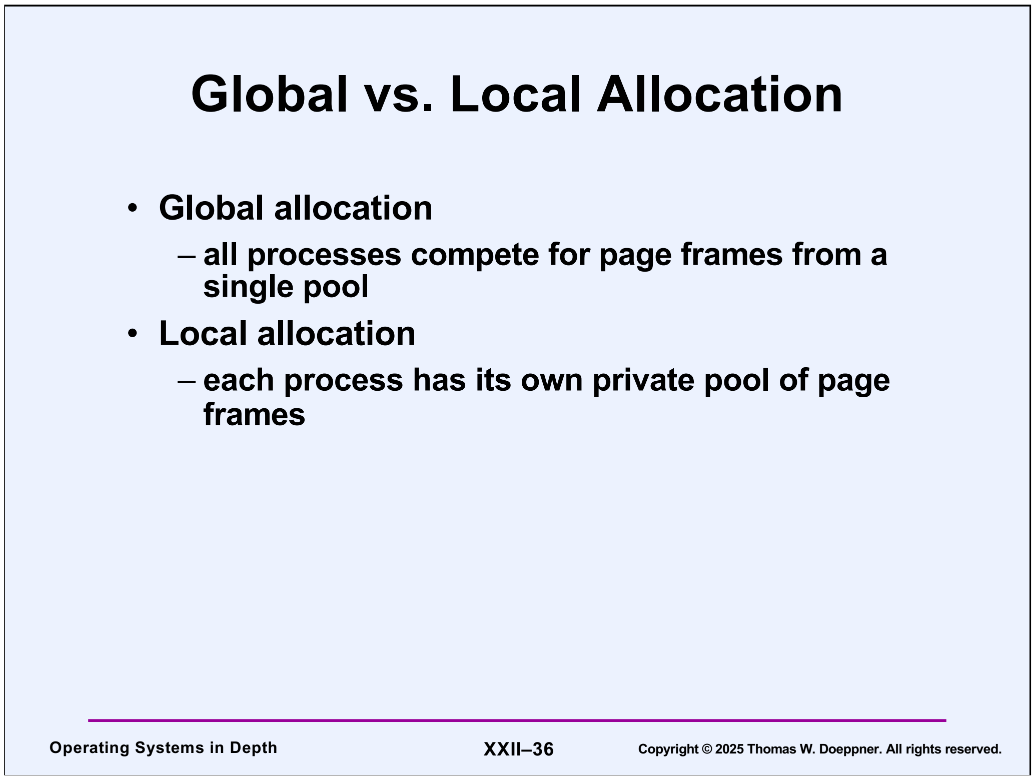
Quiz 4

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

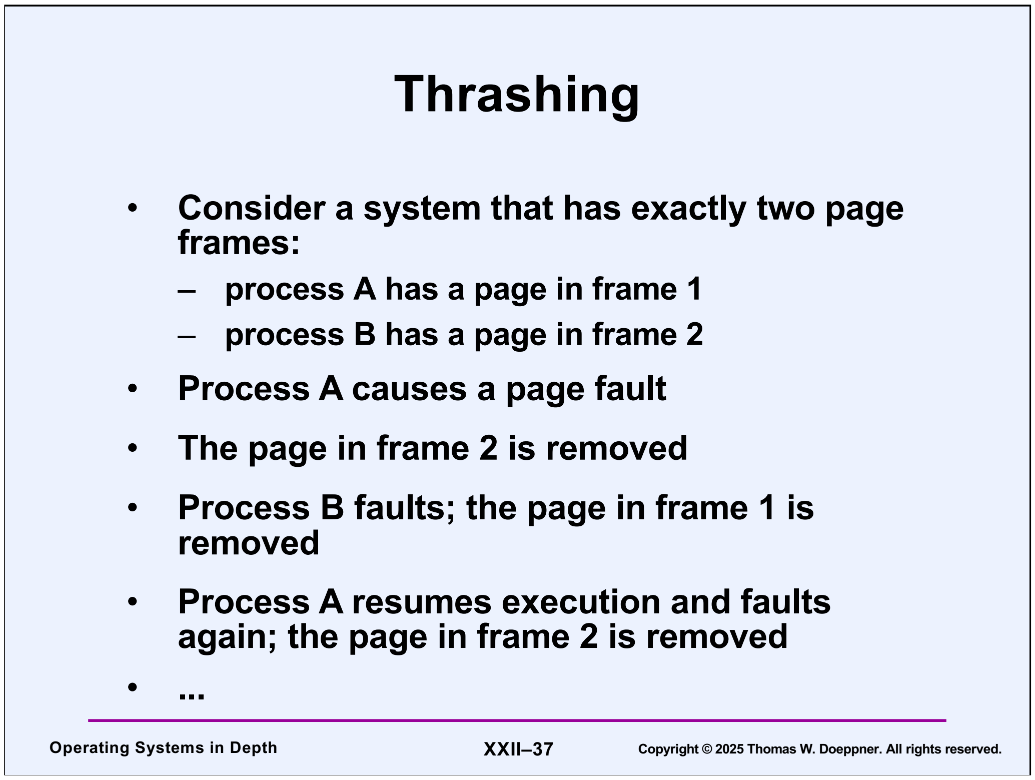


Global vs. Local Allocation

- 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.

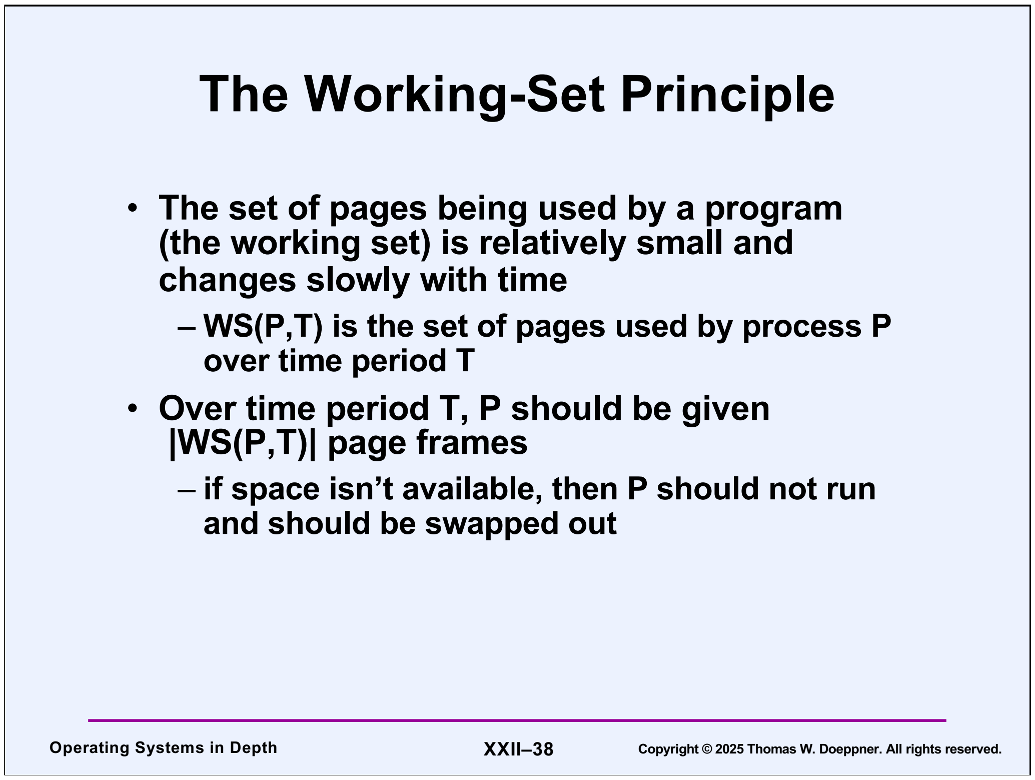


Thrashing

- 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).



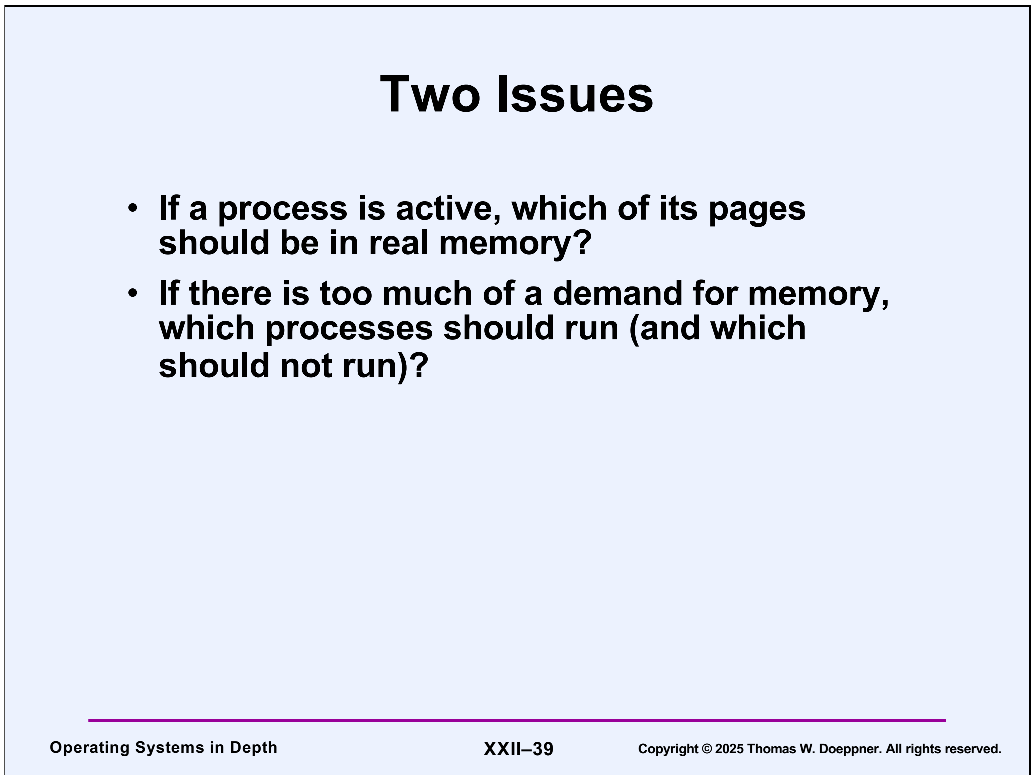
The Working-Set Principle

- 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.

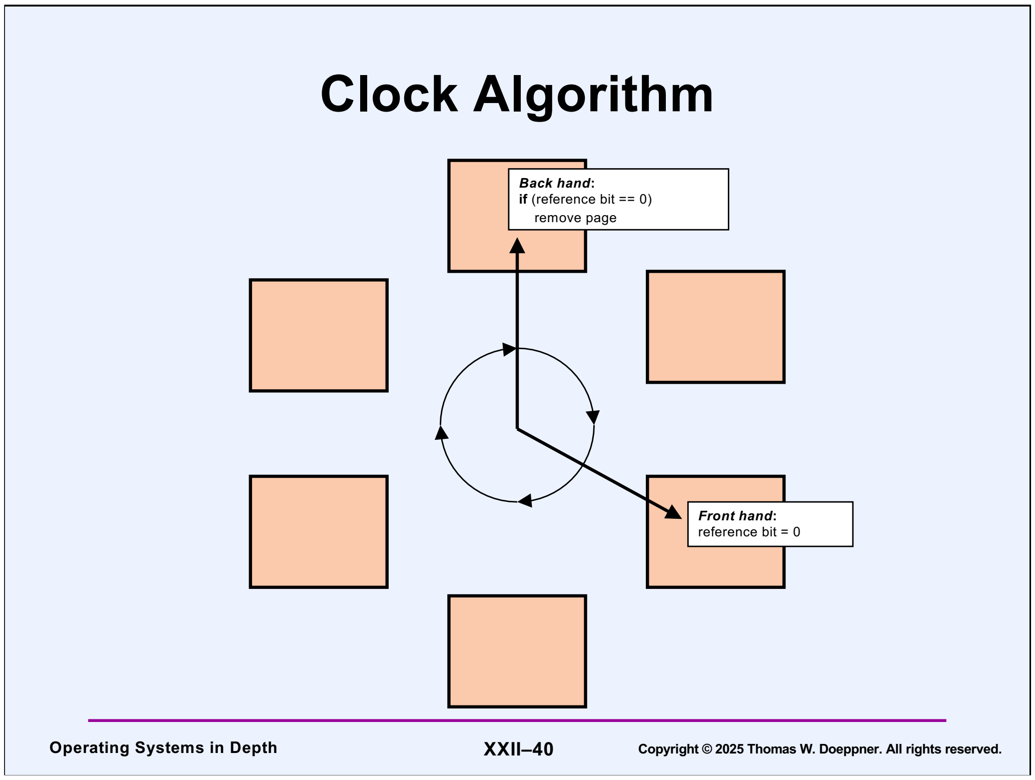
The working-set principle is used in few, if any systems (the size of the working set is just too nebulous a concept). However, it is very important as an ideal to which paging systems are compared.



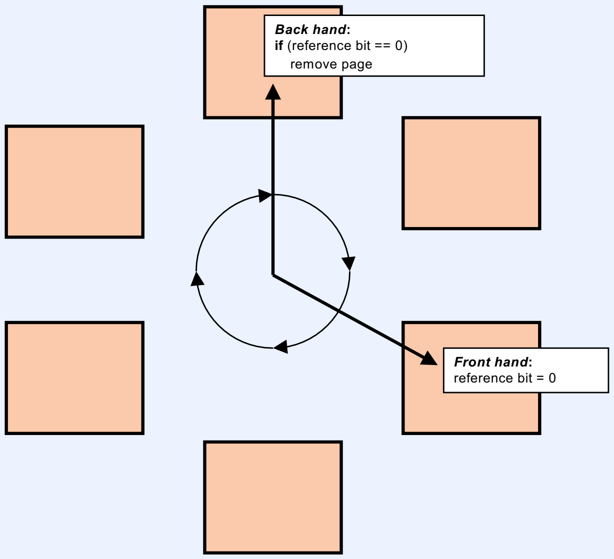
Two Issues

- 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)?

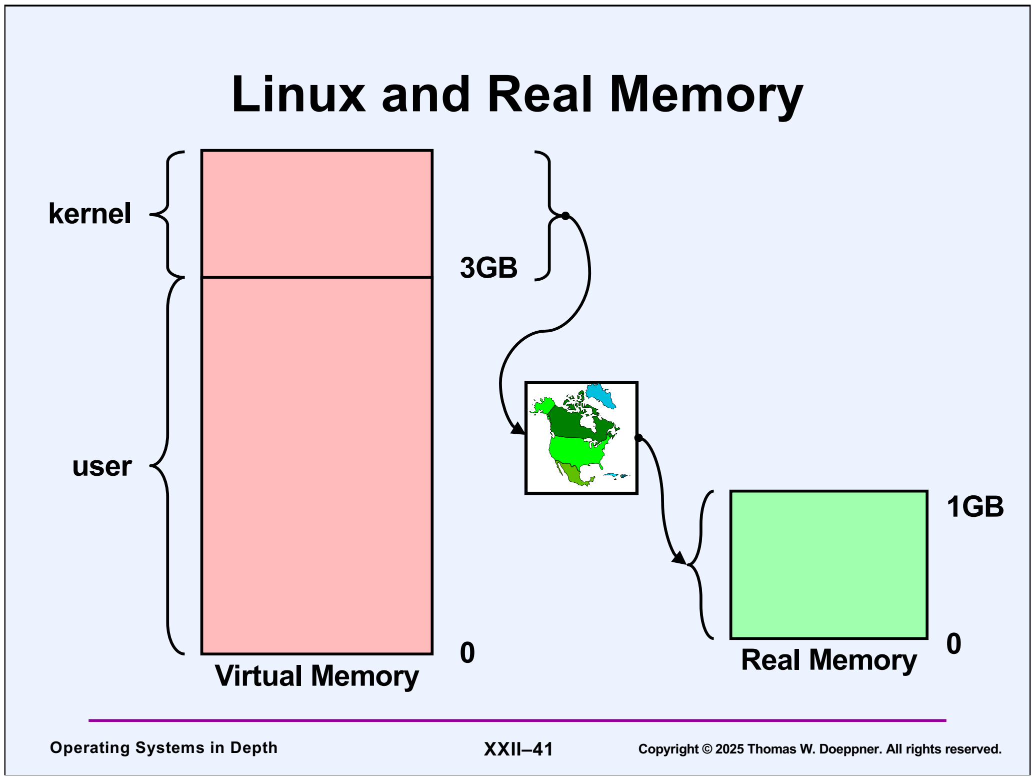
Much more attention is paid to the first issue than to the second issue.



Clock Algorithm



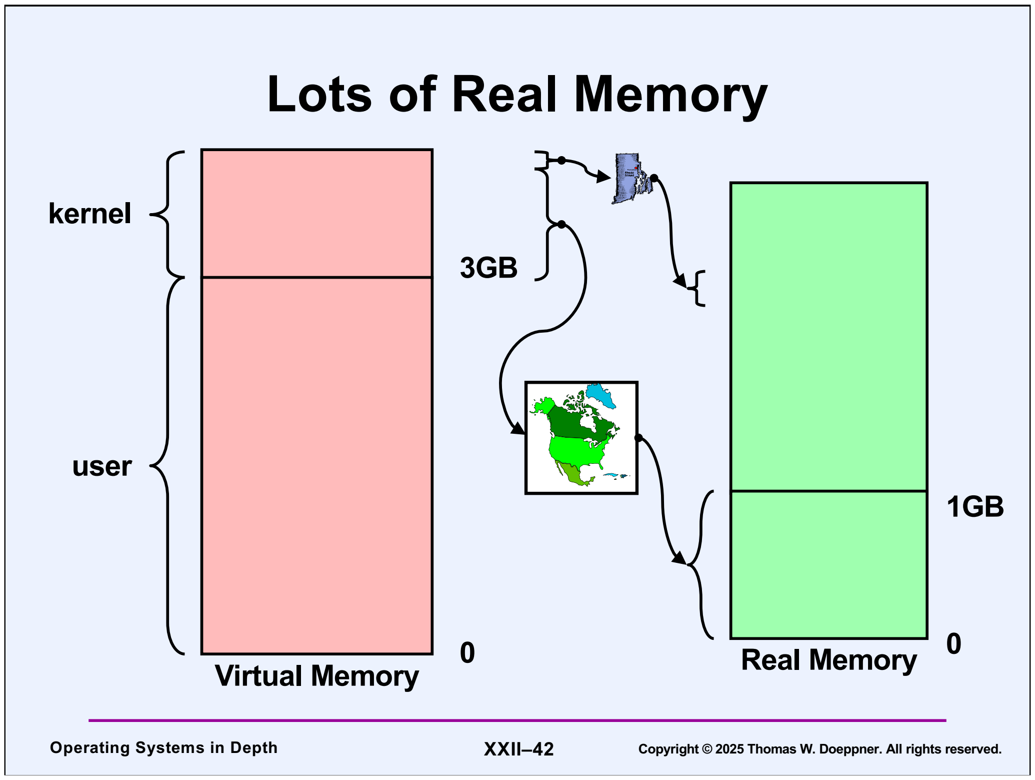
The clock algorithm is an approximation of LRU combined with LFU. As long as a page is referenced sufficiently often, it stays in memory. A primary benefit of the clock algorithm is that its easily implemented and doesn't have a high cost in terms of processor time.



Linux and Real Memory

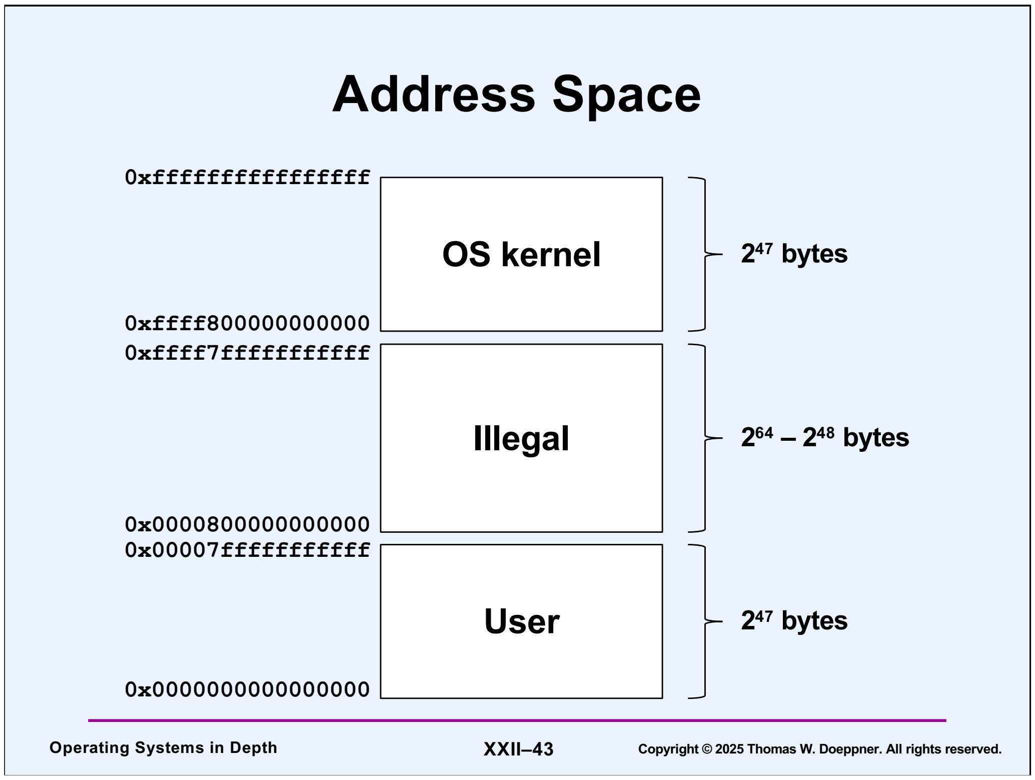
Linux, like most operating systems, doesn't separately allocate virtual memory and real memory for its own data structures—it allocates the two together. But it also must allocate real memory to user processes separately from virtual memory. The "trick" it employs is to map all of real memory into its 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.

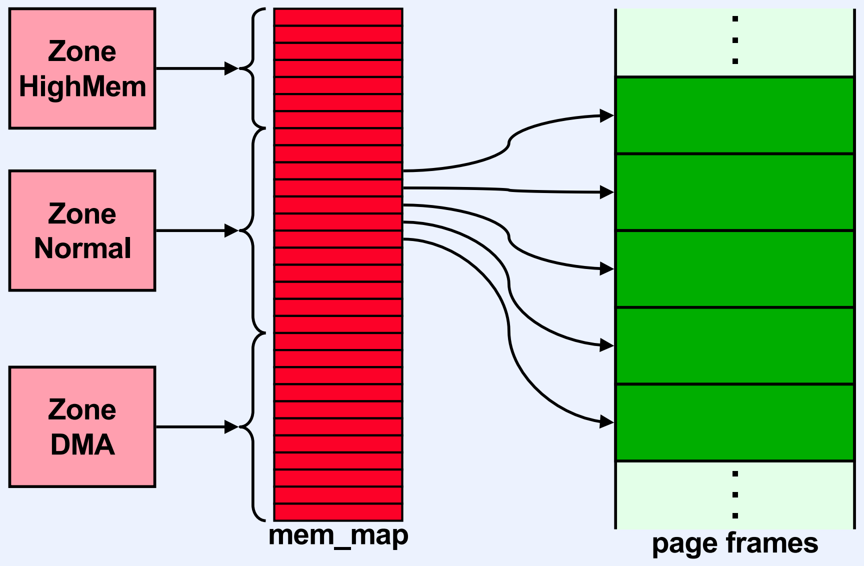


Address Space

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 5

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

b) 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

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