D. Physical memory reference

Parameter	Value
Byte offset	
Cache index	
Cache tag	
Cache hit? (Y/N)	
Cache byte returned	

9.7 Case Study: The Intel Core i7/Linux Memory System

We conclude our discussion of virtual memory mechanisms with a case study of a real system: an Intel Core i7 running Linux. Although the underlying Haswell microarchitecture allows for full 64-bit virtual and physical address spaces, the current Core i7 implementations (and those for the foreseeable future) support a 48-bit (256 TB) virtual address space and a 52-bit (4 PB) physical address space, along with a compatibility mode that supports 32-bit (4 GB) virtual and physical address spaces.

Figure 9.21 gives the highlights of the Core i7 memory system. The *processor package* (chip) includes four cores, a large L3 cache shared by all of the cores, and

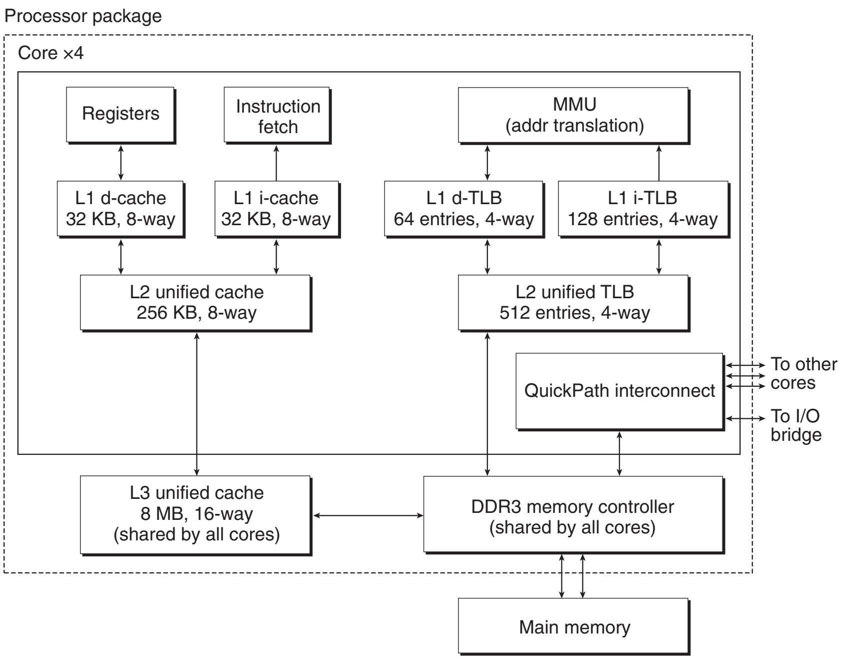

Processor package
Core ×4
Registers
Instruction fetch
MMU (addr translation)
L1 d-cache 32 KB, 8-way
L1 i-cache 32 KB, 8-way
L1 d-TLB 64 entries, 4-way
L1 i-TLB 128 entries, 4-way
L2 unified cache 256 KB, 8-way
L2 unified TLB 512 entries, 4-way
QuickPath interconnect
To other cores
To I/O bridge
L3 unified cache 8 MB, 16-way (shared by all cores)
DDR3 memory controller (shared by all cores)
Main memory

Figure 9.21 **The Core i7 memory system.**

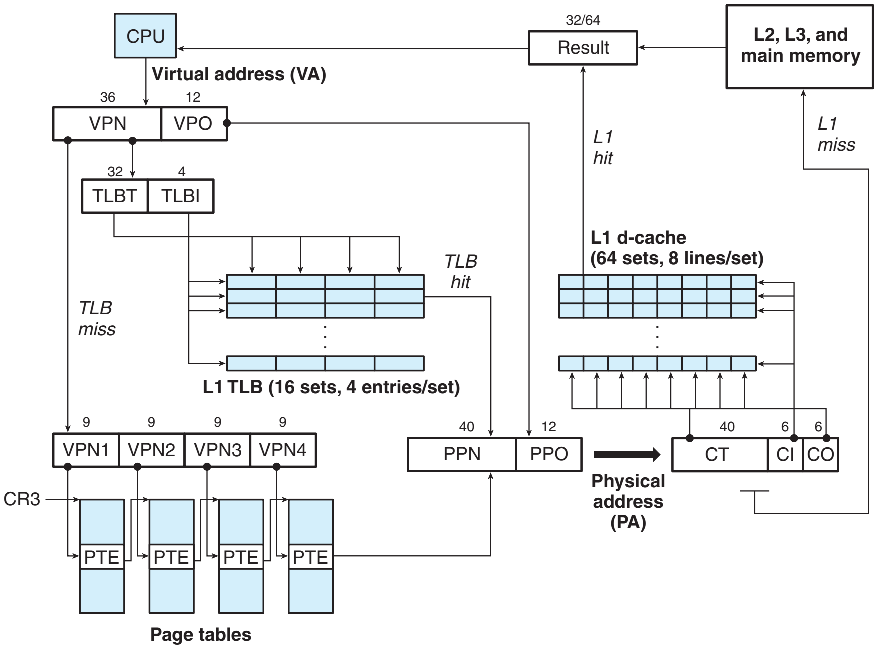
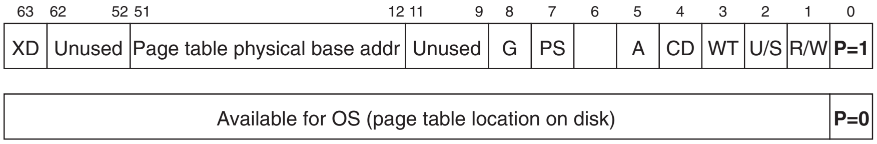


Figure 9.22 **Summary of Core i7 address translation.** For simplicity, the i-caches, i-TLB, and L2 unified TLB are not shown.

a DDR3 memory controller. Each core contains a hierarchy of TLBs, a hierarchy of data and instruction caches, and a set of fast point-to-point links, based on the QuickPath technology, for communicating directly with the other cores and the external I/O bridge. The TLBs are virtually addressed, and 4-way set associative. The L1, L2, and L3 caches are physically addressed, with a block size of 64 bytes. L1 and L2 are 8-way set associative, and L3 is 16-way set associative. The page size can be configured at start-up time as either 4 KB or 4 MB. Linux uses 4 KB pages.

9.7.1 Core i7 Address Translation

Figure 9.22 summarizes the entire Core i7 address translation process, from the time the CPU generates a virtual address until a data word arrives from memory. The Core i7 uses a four-level page table hierarchy. Each process has its own private page table hierarchy. When a Linux process is running, the page tables associated with allocated pages are all memory-resident, although the Core i7 architecture allows these page tables to be swapped in and out. The *CR3* control register contains the physical address of the beginning of the level 1 (L1) page table. The value of CR3 is part of each process context, and is restored during each context switch.


63
62
52 51
12 11
9
8
7
6
5
4
3
2
1
0
XD
Unused
Page table physical base addr
Unused
G
PS
A
CD
WT
U/S
R/W
P=1
Available for OS (page table location on disk)
P=0

Field	Description
P	Child page table present in physical memory (1) or not (0).
R/W	Read-only or read-write access permission for all reachable pages.
U/S	User or supervisor (kernel) mode access permission for all reachable pages.
WT	Write-through or write-back cache policy for the child page table.
CD	Caching disabled or enabled for the child page table.
A	Reference bit (set by MMU on reads and writes, cleared by software).
PS	Page size either 4 KB or 4 MB (defined for level 1 PTEs only).
Base addr	40 most significant bits of physical base address of child page table.
XD	Disable or enable instruction fetches from all pages reachable from this PTE.

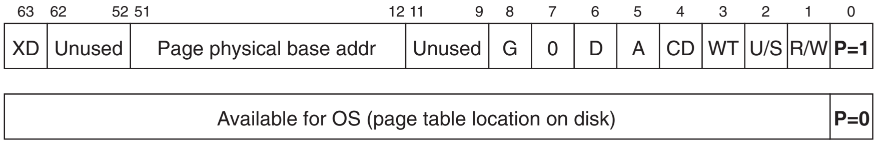
Figure 9.23 **Format of level 1, level 2, and level 3 page table entries.** Each entry references a 4 KB child page table.

Figure 9.23 shows the format of an entry in a level 1, level 2, or level 3 page table. When $P = 1$ (which is always the case with Linux), the address field contains a 40-bit physical page number (PPN) that points to the beginning of the appropriate page table. Notice that this imposes a 4 KB alignment requirement on page tables.

Figure 9.24 shows the format of an entry in a level 4 page table. When $P = 1$, the address field contains a 40-bit PPN that points to the base of some page in physical memory. Again, this imposes a 4 KB alignment requirement on physical pages.

The PTE has three permission bits that control access to the page. The R/W bit determines whether the contents of a page are read/write or read-only. The U/S bit, which determines whether the page can be accessed in user mode, protects code and data in the operating system kernel from user programs. The XD (execute disable) bit, which was introduced in 64-bit systems, can be used to disable instruction fetches from individual memory pages. This is an important new feature that allows the operating system kernel to reduce the risk of buffer overflow attacks by restricting execution to the read-only code segment.

As the MMU translates each virtual address, it also updates two other bits that can be used by the kernel's page fault handler. The MMU sets the A bit, which is known as a *reference bit*, each time a page is accessed. The kernel can use the reference bit to implement its page replacement algorithm. The MMU sets the D bit, or *dirty bit*, each time the page is written to. A page that has been modified is sometimes called a *dirty page*. The dirty bit tells the kernel whether or not it must



Field	Description
P	Child page present in physical memory (1) or not (0).
R/W	Read-only or read/write access permission for child page.
U/S	User or supervisor mode (kernel mode) access permission for child page.
WT	Write-through or write-back cache policy for the child page.
CD	Cache disabled or enabled.
A	Reference bit (set by MMU on reads and writes, cleared by software).
D	Dirty bit (set by MMU on writes, cleared by software).
G	Global page (don't evict from TLB on task switch).
Base addr	40 most significant bits of physical base address of child page.
XD	Disable or enable instruction fetches from the child page.

Figure 9.24 **Format of level 4 page table entries.** Each entry references a 4 KB child page.

write back a victim page before it copies in a replacement page. The kernel can call a special kernel-mode instruction to clear the reference or dirty bits.

Figure 9.25 shows how the Core i7 MMU uses the four levels of page tables to translate a virtual address to a physical address. The 36-bit VPN is partitioned into four 9-bit chunks, each of which is used as an offset into a page table. The CR3 register contains the physical address of the L1 page table. VPN 1 provides an offset to an L1 PTE, which contains the base address of the L2 page table. VPN 2 provides an offset to an L2 PTE, and so on.

9.7.2 Linux Virtual Memory System

A virtual memory system requires close cooperation between the hardware and the kernel. Details vary from version to version, and a complete description is beyond our scope. Nonetheless, our aim in this section is to describe enough of the Linux virtual memory system to give you a sense of how a real operating system organizes virtual memory and how it handles page faults.

Linux maintains a separate virtual address space for each process of the form shown in Figure 9.26. We have seen this picture a number of times already, with its familiar code, data, heap, shared library, and stack segments. Now that we understand address translation, we can fill in some more details about the kernel virtual memory that lies above the user stack.

The kernel virtual memory contains the code and data structures in the kernel. Some regions of the kernel virtual memory are mapped to physical pages that

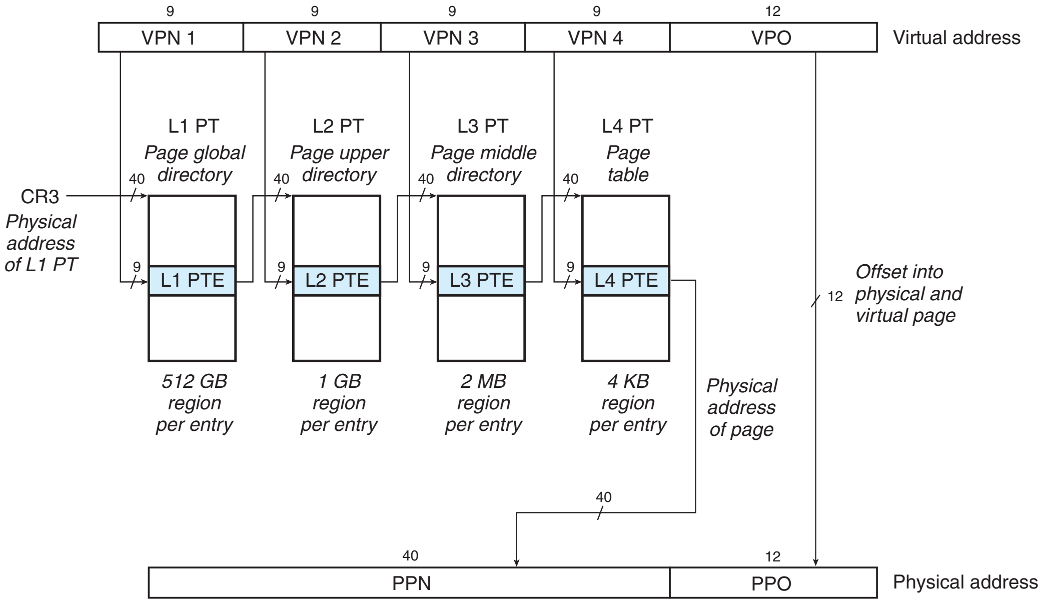
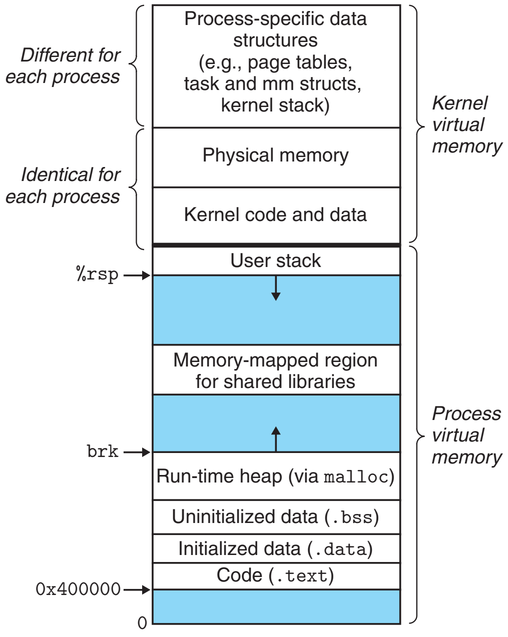


Figure 9.25 **Core i7 page table translation.** PT: page table; PTE: page table entry; VPN: virtual page number; VPO: virtual page offset; PPN: physical page number; PPO: physical page offset. The Linux names for the four levels of page tables are also shown.

Figure 9.26
The virtual memory of a Linux process.


Process-specific data structures (e.g., page tables, task and mm structs, kernel stack)
Physical memory
Kernel code and data
Different for each process
Identical for each process
Kernel virtual memory
User stack
%rsp
Memory-mapped region for shared libraries
brk
Run-time heap (via malloc)
Uninitialized data (.bss)
Initialized data (.data)
Code (.text)
0x400000
0
Process virtual memory

Aside Optimizing address translation

In our discussion of address translation, we have described a sequential two-step process where the MMU (1) translates the virtual address to a physical address and then (2) passes the physical address to the L1 cache. However, real hardware implementations use a neat trick that allows these steps to be partially overlapped, thus speeding up accesses to the L1 cache. For example, a virtual address on a Core i7 with 4 KB pages has 12 bits of VPO, and these bits are identical to the 12 bits of PPO in the corresponding physical address. Since the 8-way set associative physically addressed L1 caches have 64 sets and 64-byte cache blocks, each physical address has 6 ($\log_2 64$) cache offset bits and 6 ($\log_2 64$) index bits. These 12 bits fit exactly in the 12-bit VPO of a virtual address, which is no accident! When the CPU needs a virtual address translated, it sends the VPN to the MMU and the VPO to the L1 cache. While the MMU is requesting a page table entry from the TLB, the L1 cache is busy using the VPO bits to find the appropriate set and read out the eight tags and corresponding data words in that set. When the MMU gets the PPN back from the TLB, the cache is ready to try to match the PPN to one of these eight tags.

are shared by all processes. For example, each process shares the kernel's code and global data structures. Interestingly, Linux also maps a set of contiguous virtual pages (equal in size to the total amount of DRAM in the system) to the corresponding set of contiguous physical pages. This provides the kernel with a convenient way to access any specific location in physical memory—for example, when it needs to access page tables or to perform memory-mapped I/O operations on devices that are mapped to particular physical memory locations.

Other regions of kernel virtual memory contain data that differ for each process. Examples include page tables, the stack that the kernel uses when it is executing code in the context of the process, and various data structures that keep track of the current organization of the virtual address space.

Linux Virtual Memory Areas

Linux organizes the virtual memory as a collection of *areas* (also called *segments*). An area is a contiguous chunk of existing (allocated) virtual memory whose pages are related in some way. For example, the code segment, data segment, heap, shared library segment, and user stack are all distinct areas. Each existing virtual page is contained in some area, and any virtual page that is not part of some area does not exist and cannot be referenced by the process. The notion of an area is important because it allows the virtual address space to have gaps. The kernel does not keep track of virtual pages that do not exist, and such pages do not consume any additional resources in memory, on disk, or in the kernel itself.

Figure 9.27 highlights the kernel data structures that keep track of the virtual memory areas in a process. The kernel maintains a distinct task structure (`task_struct` in the source code) for each process in the system. The elements of the task structure either contain or point to all of the information that the kernel needs to

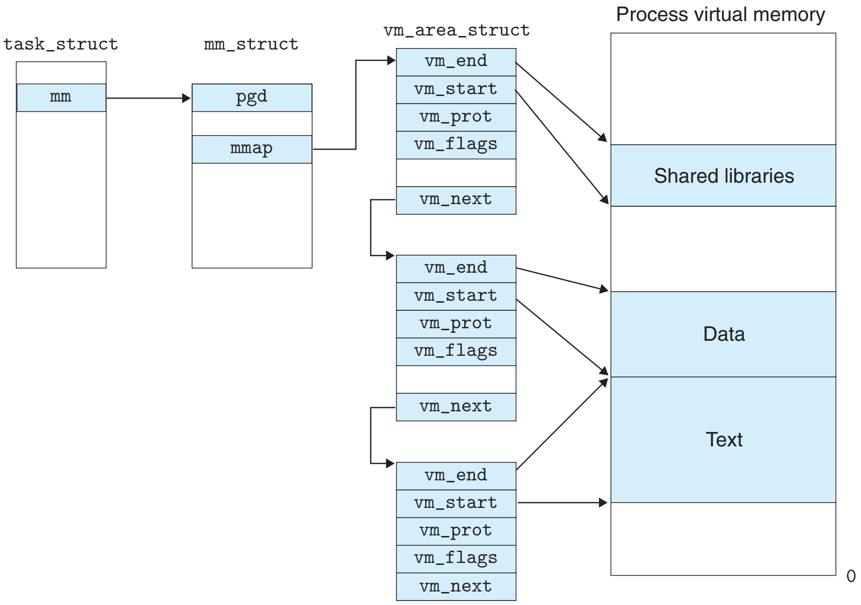


Figure 9.27 **How Linux organizes virtual memory.**

run the process (e.g., the PID, pointer to the user stack, name of the executable object file, and program counter).

One of the entries in the task structure points to an mm_struct that characterizes the current state of the virtual memory. The two fields of interest to us are pgd, which points to the base of the level 1 table (the page global directory), and mmap, which points to a list of vm_area_structs (area structs), each of which characterizes an area of the current virtual address space. When the kernel runs this process, it stores pgd in the CR3 control register.

For our purposes, the area struct for a particular area contains the following fields:

fvm_start. Points to the beginning of the area.

vm_end. Points to the end of the area.

vm_prot. Describes the read/write permissions for all of the pages contained in the area.

vm_flags. Describes (among other things) whether the pages in the area are shared with other processes or private to this process.

vm_next. Points to the next area struct in the list.

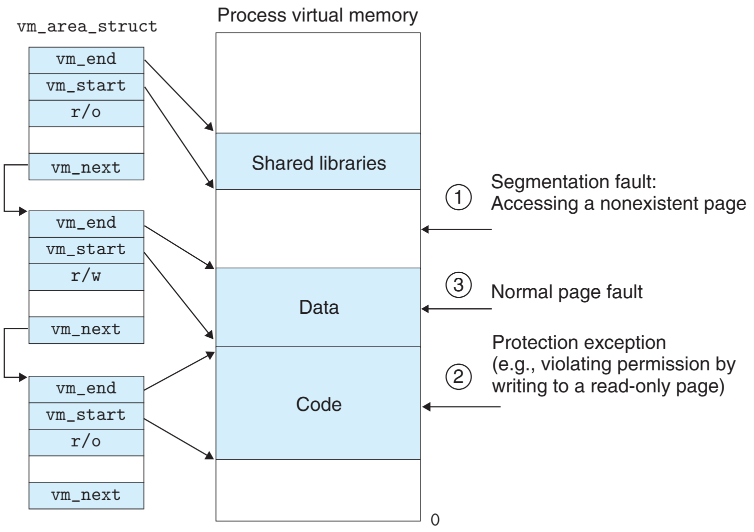


Figure 9.28
Linux page fault handling.

Linux Page Fault Exception Handling

Suppose the MMU triggers a page fault while trying to translate some virtual address *A*. The exception results in a transfer of control to the kernel's page fault handler, which then performs the following steps:

1. Is virtual address *A* legal? In other words, does *A* lie within an area defined by some area struct? To answer this question, the fault handler searches the list of area structs, comparing *A* with the `vm_start` and `vm_end` in each area struct. If the instruction is not legal, then the fault handler triggers a segmentation fault, which terminates the process. This situation is labeled "1" in Figure 9.28.

   Because a process can create an arbitrary number of new virtual memory areas (using the `mmap` function described in the next section), a sequential search of the list of area structs might be very costly. So in practice, Linux superimposes a tree on the list, using some fields that we have not shown, and performs the search on this tree.
2. Is the attempted memory access legal? In other words, does the process have permission to read, write, or execute the pages in this area? For example, was the page fault the result of a store instruction trying to write to a read-only page in the code segment? Is the page fault the result of a process running in user mode that is attempting to read a word from kernel virtual memory? If the attempted access is not legal, then the fault handler triggers a protection exception, which terminates the process. This situation is labeled "2" in Figure 9.28.
3. At this point, the kernel knows that the page fault resulted from a legal operation on a legal virtual address. It handles the fault by selecting a victim page, swapping out the victim page if it is dirty, swapping in the new page,

and updating the page table. When the page fault handler returns, the CPU restarts the faulting instruction, which sends *A* to the MMU again. This time, the MMU translates *A* normally, without generating a page fault.

9.8 Memory Mapping

Linux initializes the contents of a virtual memory area by associating it with an *object* on disk, a process known as *memory mapping*. Areas can be mapped to one of two types of objects:

1. *Regular file in the Linux file system:* An area can be mapped to a contiguous section of a regular disk file, such as an executable object file. The file section is divided into page-size pieces, with each piece containing the initial contents of a virtual page. Because of demand paging, none of these virtual pages is actually swapped into physical memory until the CPU first *touches* the page (i.e., issues a virtual address that falls within that page's region of the address space). If the area is larger than the file section, then the area is padded with zeros.
2. *Anonymous file:* An area can also be mapped to an anonymous file, created by the kernel, that contains all binary zeros. The first time the CPU touches a virtual page in such an area, the kernel finds an appropriate victim page in physical memory, swaps out the victim page if it is dirty, overwrites the victim page with binary zeros, and updates the page table to mark the page as resident. Notice that no data are actually transferred between disk and memory. For this reason, pages in areas that are mapped to anonymous files are sometimes called *demand-zero pages*.

In either case, once a virtual page is initialized, it is swapped back and forth between a special *swap file* maintained by the kernel. The swap file is also known as the *swap space* or the *swap area*. An important point to realize is that at any point in time, the swap space bounds the total amount of virtual pages that can be allocated by the currently running processes.

9.8.1 Shared Objects Revisited

The idea of memory mapping resulted from a clever insight that if the virtual memory system could be integrated into the conventional file system, then it could provide a simple and efficient way to load programs and data into memory.

As we have seen, the process abstraction promises to provide each process with its own private virtual address space that is protected from errant writes or reads by other processes. However, many processes have identical read-only code areas. For example, each process that runs the Linux shell program `bash` has the same code area. Further, many programs need to access identical copies of read-only run-time library code. For example, every C program requires functions from the standard C library such as `printf`. It would be extremely wasteful for each process to keep duplicate copies of these commonly used codes in physical