5.7 Virtual memory

> "… a system has been devised to make the core drum combination appear to the programmer as a single level store, the requisite transfers taking place automatically.
> *Kilburn et al., One-level storage system, 1962.*

In earlier sections, we saw how caches provided fast access to recently-used portions of a program's code and data. Similarly, the main memory can act as a "cache" for the secondary storage, traditionally implemented with magnetic disks. This technique is called *virtual memory*. Historically, there were two major motivations for virtual memory: to allow efficient and safe sharing of memory among several programs, such as for the memory needed by multiple virtual machines for Cloud computing, and to remove the programming burdens of a small, limited amount of main memory. Five decades after its invention, it's the former reason that reigns today.

Virtual memory: A technique that uses main memory as a "cache" for secondary storage.

Of course, to allow multiple virtual machines to share the same memory, we must be able to protect the virtual machines from each other, ensuring that a program can just read and write the portions of main memory that have been assigned to it. Main memory need contain only the active portions of the many virtual machines, just as a cache contains only the active portion of one program. Thus, the principle of locality enables virtual memory as well as caches, and virtual memory allows us to share the processor efficiently as well as the main memory.

We cannot know which virtual machines will share the memory with other virtual machines when we compile them. In fact, the virtual machines sharing the memory change dynamically while they are running. Because of this dynamic interaction, we would like to compile each program into its own *address space*—a separate range of memory locations accessible only to this program. Virtual memory implements the translation of a program's address space to *physical addresses*. This translation process enforces *protection* of a program's address space from other virtual machines.

Physical address: An address in main memory.

Protection: A set of mechanisms for ensuring that multiple processes sharing the processor, memory, or I/O devices cannot interfere, intentionally or unintentionally, with one another by reading or writing each other's data. These mechanisms also isolate the operating system from a user process.

The second motivation for virtual memory is to allow a single user program to exceed the size of primary memory. Formerly, if a program became too large for memory, it was up to the programmer to make it fit. Programmers divided programs into pieces and then identified the pieces that were mutually exclusive. These overlays were loaded or unloaded under user program control during execution, with the programmer ensuring that the program at no time tried to access an overlay that was not loaded and that the overlays loaded never exceeded the total size of the memory. Overlays were traditionally organized as modules, each containing both code and data. Calls between procedures in different modules would lead to overlaying of one module with another.

As you can well imagine, this responsibility was a substantial burden on programmers. Virtual memory, which was invented to relieve programmers of this difficulty, automatically manages the two levels of the memory hierarchy represented by main memory (sometimes called physical memory to distinguish it from virtual memory) and secondary storage.

Although the concepts at work in virtual memory and in caches are the same, their differing historical roots have led to the use of different terminology. A virtual memory block is called a *page*, and a virtual memory miss is called a *page fault*. With virtual memory, the processor produces a *virtual address*, which is translated by a combination of hardware and software to a physical address, which in turn can be used to access main memory. The figure below shows the virtually addressed memory with pages mapped to main memory. This process is called *address mapping* or *address translation*. Today, the two memory hierarchy levels controlled by virtual memory are usually DRAMs and flash memory in personal mobile devices and DRAMs and magnetic disks in servers (see COD Section 5.2 (Memory technologies)). If we return to our library analogy, we can think of a virtual address as the title of a book and a physical address as the location of that book in the library, such as might be given by the Library of Congress call number.

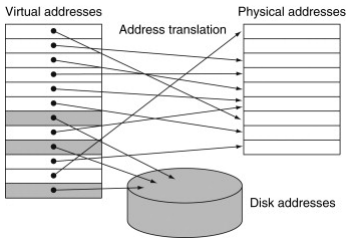
Page fault: An event that occurs when an accessed page is not present in main memory.

Virtual address: An address that corresponds to a location in virtual space and is translated by address mapping to a physical address when memory is accessed.

Address translation: Also called ***address mapping***. The process by which a virtual address is mapped to an address used to access memory.

Figure 5.7.1: In virtual memory, blocks of memory (called pages) are mapped from one set of addresses (called virtual addresses) to another set (called physical addresses) (COD Figure 5.25).

The processor generates virtual addresses while the memory is accessed using physical addresses. Both the virtual memory and the physical memory are broken into pages, so that a virtual page is mapped to a physical page. Of course, it is also possible for a virtual page to be absent from main memory and not be mapped to a physical address; in that case, the page resides on disk. Physical pages can be shared by having two virtual addresses point to the same physical address. This capability is used to allow two different programs to share data or code.



PARTICIPATION ACTIVITY 5.7.1: Virtual memory basics.

Protection | Page fault | Virtual address | Virtual memory | Address mapping

Mechanisms that prevent multiple processes that use the same hardware from interfering with each other.

A technique where main memory is used as a cache for secondary storage.

The process of mapping a virtual address to a physical address.

An address that corresponds to a location in virtual space.

A virtual memory miss.

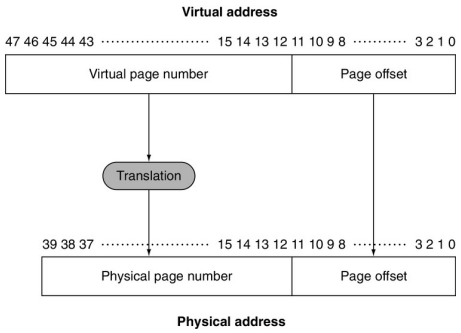
Reset

Virtual memory also simplifies loading the program for execution by providing *relocation*. Relocation maps the virtual addresses used by a program to different physical addresses before the addresses are used to access memory. This relocation allows us to load the program anywhere in main memory. Furthermore, all virtual memory systems in use today relocate the program as a set of fixed-size blocks (pages), thereby eliminating the need to find a contiguous block of memory to allocate to a program; instead, the operating system needs only to find enough pages in main memory.

In virtual memory, the address is broken into a *virtual page number* and a *page offset*. The figure below shows the translation of the virtual page number to a *physical page number*. While ARMv8 has a 64-bit address, the upper 16 bits are not used, so the address to be mapped is 48 bits. This figure assumes the physical memory is 1 TiB, or 2^{40} bytes, which needs a 40-bit address. The physical page number constitutes the upper portion of the physical address, while the page offset, which is not changed, constitutes the lower portion. The number of bits in the page offset field determines the page size. The number of pages addressable with the virtual address can be different than the number of pages addressable with the physical address. Having a larger number of virtual pages than physical pages is the basis for the illusion of an essentially unbounded amount of virtual memory.

Figure 5.7.2: Mapping from a virtual to a physical address (COD Figure 5.26).

The page size is 2^{12} = 4 KiB. The number of physical pages allowed in memory is 2^{28}, since the physical page number has 28 bits in it. Thus, main memory can have at most 1 TiB, while the virtual address space is 256 TiB. ARMv8 allows physical memory to be up to 256 TiB; we chose 1 TiB since that matches the maximum of some ARMv8 computers in 2015.



Many design choices in virtual memory systems are motivated by the high cost of a page fault. A page fault to disk will take millions of clock cycles to process. (The table in COD Section 5.2 (Memory technologies) shows that main memory latency is about 100,000 times quicker than disk.) This enormous miss penalty, dominated by the time to get the first word for typical page sizes, leads to several key decisions in designing virtual memory systems:

- Pages should be large enough to try to amortize the high access time. Sizes from 4 KiB to 64 KiB are typical today. New desktop and server systems are being developed to support 32 KiB and 64 KiB pages, but new embedded systems are going in the other direction, to 1 KiB pages.
- Organizations that reduce the page fault rate are attractive. The primary technique used here is to allow fully associative placement of pages in memory.
- Page faults can be handled in software because the overhead will be small compared to the disk access time. In addition, software can afford to use clever algorithms for choosing how to place pages because even little reductions in the miss rate will pay for the cost of such algorithms.

- Write-through will not work for virtual memory, since writes take too long. Instead, virtual memory systems use write-back.

The next few subsections address these factors in virtual memory design.

Elaboration

We present the motivation for virtual memory as many virtual machines sharing the same memory, but virtual memory was originally invented so that many programs could share a computer as part of a timesharing system. Since many readers today have no experience with time-sharing systems, we use virtual machines to motivate this section.

Elaboration

ARMv8 uses the term granule instead of page, and it calls a page fault a Memory Management Unit (MMU) exception. While the maximum virtual address is 48 bits, ARMv8 allows implementations with a smaller virtual address. It also allows physical addresses as large as 48 bits. It supports three options for minimum page or granule size: 4, 16, and 64 Kibibyte.

Elaboration

For servers and even PCs, 32-bit address processors are problematic. Although we normally think of virtual addresses as much larger than physical addresses, the opposite can occur when the processor address size is small relative to the state of the memory technology. No single program or virtual machine can benefit, but a collection of programs or virtual machines running at the same time can benefit from not having to be swapped out of main memory or by running on parallel processors.

Elaboration

The discussion of virtual memory in this book focuses on paging, which uses fixed-size blocks. There is also a variable-size block scheme called segmentation. In segmentation, an address consists of two parts: a segment number and a segment offset. The segment number is mapped to a physical address, and the offset is added to find the actual physical address. Because the segment can vary in size, a bounds check is also needed to make sure that the offset is within the segment. The major use of segmentation is to support more powerful methods of protection and sharing in an address space. Most operating system textbooks contain extensive discussions of segmentation compared to paging and of the use of segmentation to to share the address space logically. The major disadvantage of segmentation is that it splits the address space into logically separate pieces that must be manipulated as a two-part address: the segment number and the offset. Paging, in contrast, makes the boundary between page number and offset invisible to programmers and compilers.

Segments have also been used as a method to extend the address space without changing the word size of the computer. Such attempts have been unsuccessful because of the awkwardness and performance penalties inherent in a two-part address, of which programmers and compilers must be aware.

Many architectures divide the address space into large fixed-size blocks that simplify protection between the operating system and user programs and increase the efficiency of implementing paging. Although these divisions are often called "segments," this mechanism is much simpler than variable block size segmentation and is not visible to user programs; we discuss it in more detail shortly.

Segmentation: A variable-size address mapping scheme in which an address consists of two parts: a segment number, which is mapped to a physical address, and a segment offset.

Placing a page and finding it again

Because of the incredibly high penalty for a page fault, designers reduce page fault frequency by optimizing page placement. If we allow a virtual page to be mapped to any physical page, the operating system can then choose to replace any page it wants when a page fault occurs. For example, the operating system can use a sophisticated algorithm and complex data structures that track page usage to try to choose a page that will not be needed for a long time. The ability to use a clever and flexible replacement scheme reduces the page fault rate and simplifies the use of fully associative placement of pages.

As mentioned in COD Section 5.4 (Measuring and improving cache performance), the difficulty in using fully associative placement is in locating an entry, since it can be anywhere in the upper level of the hierarchy. A full search is impractical. In virtual memory systems, we locate pages by using a table that indexes the main memory; this structure is called a *page table*, and it resides in main memory. A page table is indexed by the page number from the virtual address to discover the corresponding physical page number. Each program has its own page table, which maps the virtual address space of that program to main memory. In our library analogy, the page table corresponds to a mapping between book titles and library locations. Just as the card catalog may contain entries for books in another library on campus rather than the local branch library, we will see that the page table may contain entries for pages not present in memory. To indicate the location of the page table in memory, the hardware includes a register that points to the start of the page table; we call this the page table register. Assume for now that the page table is in a fixed and contiguous area of memory.

Page table: The table containing the virtual to physical address translations in a virtual memory system. The table, which is stored in memory, is typically indexed by the virtual page number; each entry in the table contains the physical page number for that virtual page if the page is currently in memory.

Hardware/Software Interface

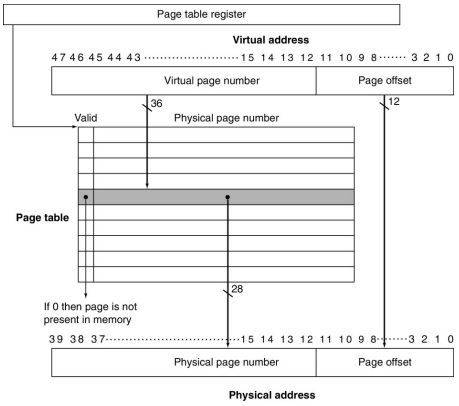
The page table, together with the program counter and the registers, specifies the *state* of a virtual machine. If we want to allow another virtual machine to use the processor, we must save this state. Later, after restoring this state, the virtual machine can continue execution. We often refer to this state as a *process*. The process is considered *active* when it is in possession of the processor; otherwise, it is considered *inactive*. The operating system can make a process active by loading the process's state, including the program counter, which will initiate execution at the value of the saved program counter.

The process's address space, and hence all the data it can access in memory, is defined by its page table, which resides in memory. Rather than save the entire page table, the operating system simply loads the page table register to point to the page table of the process it wants to make active. Each process has its own page table, since different processes use the same virtual addresses. The operating system is responsible for allocating the physical memory and updating the page tables, so that the virtual address spaces of distinct processes do not collide. As we will see shortly, the use of separate page tables also provides protection of one process from another.

The figure below uses the page table register, the virtual address, and the indicated page table to show how the hardware can form a physical address. A valid bit is used in each page table entry, just as we did in a cache. If the bit is off, the page is not present in main memory and a page fault occurs. If the bit is on, the page is in memory and the entry contains the physical page number.

Figure 5.7.3: The page table is indexed with the virtual page number to obtain the corresponding portion of the physical address (COD Figure 5.27).

We assume a 48-bit address. The page table pointer gives the starting address of the page table. In this figure, the page size is 2^{12} bytes, or 4 KiB. The virtual address space is 2^{48} bytes, or 256 TiB, and the physical address space is 2^{40} bytes, which allows main memory of up to 1 TiB. If ARMv8 used a single page table as shown in this figure, the number of entries in the page table would be 2^{36}, or about 64 billion entries. (We'll see what ARMv8 does to reduce the number of entries shortly.) The valid bit for each entry indicates whether the mapping is legal. If it is off, then the page is not present in memory. Although the page table entry shown here need only be 29 bits wide, it would typically be rounded up to a power of 2 bits for ease of indexing. The page table entries in ARMv8 are 64 bits. The extra bits would be used to store additional information that needs to be kept on a per-page basis, such as protection.



Because the page table contains a mapping for every possible virtual page, no tags are required. In cache terminology, the index that is used to access the page table consists of the full block address, which in this case is the virtual page number.

PARTICIPATION ACTIVITY 5.7.2: Page tables.

1) Each program has a page table.
   - ○ True
   - ○ False

2) In a page table, a physical address is located by indexing a virtual page number.
   - ○ True
   - ○ False

3) The virtual address indicates the location of the page table in memory.
   - ○ True
   - ○ False

Page faults

If the valid bit for a virtual page is off, a page fault occurs. The operating system must be given control. This transfer is done with the exception mechanism, which we saw in COD Chapter 4 (The Processor) and will discuss again later in this section. Once the operating system gets control, it must find the page in the next level of the hierarchy (usually flash memory or magnetic disk) and decide where to place the requested page in the main memory.

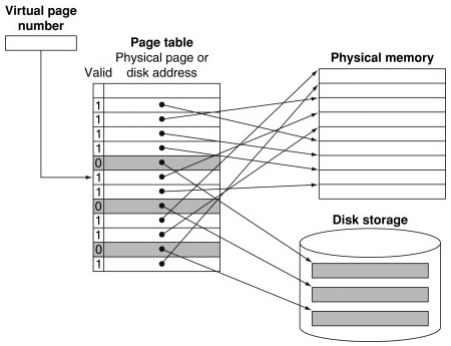
The virtual address alone does not immediately tell us where the page is in secondary memory. Returning to our library analogy, we cannot find the location of a library book on the shelves just by knowing its title. Instead, we go to the catalog and look up the book, obtaining an address for the location on the shelves, such as the Library of Congress call number. Likewise, in a virtual memory system, we must keep track of the location in secondary memory of each page in virtual address space.

Because we do not know ahead of time when a page in memory will be replaced, the operating system usually creates the space on flash memory or disk for all the pages of a process when it creates the process. This space is called the *swap space*. At that time, it also creates a data structure to record where each virtual page is stored on disk. This data structure may be part of the page table or may be an auxiliary data structure indexed in the same way as the page table. The figure below shows the organization when a single table holds either the physical page number or the secondary memory address.

Swap space: The space on the disk reserved for the full virtual memory space of a process.

Figure 5.7.4: The page table maps each page in virtual memory to either a page in main memory or a page stored on disk, which is the next level in the hierarchy (COD Figure 5.28).

The virtual page number is used to index the page table. If the valid bit is on, the page table supplies the physical page number (i.e., the starting address of the page in memory) corresponding to the virtual page. If the valid bit is off, the page currently resides only on disk, at a specified disk address. In many systems, the table of physical page addresses and disk page addresses, while logically one table, is stored in two separate data structures. Dual tables are justified in part because we must keep the disk addresses of all the pages, even if they are currently in main memory. Remember that the pages in main memory and the pages on disk are the same size.



The operating system also creates a data structure that tracks which processes and which virtual addresses use each physical page. When a page fault occurs, if all the pages in main memory are in use, the operating system must choose a page to replace. Because we want to minimize the number of page faults, most operating systems try to choose a page that they hypothesize will not be needed soon. Using the past to predict the future, operating systems follow the *least recently used* (LRU) replacement scheme, which we mentioned in COD Section 5.4 (Measuring and improving cache performance). The operating system searches for the least recently used page, assuming that a page that has not been used in a long time is less likely to be needed than a more recently accessed page. The replaced pages are written to swap space in secondary memory. In case you are wondering, the operating system is just another process, and these tables controlling memory are in memory; the details of this seeming contradiction will be explained shortly.

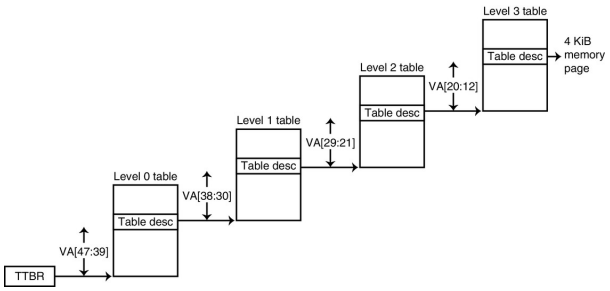
Hardware/Software Interface

Implementing a completely accurate LRU scheme is too expensive, since it requires updating a data structure on every memory reference. Thus, most operating systems approximate LRU by keeping track of which pages have and which pages have not been recently used. To help the operating system estimate the LRU pages, some computers provide a *reference bit* or *use bit*, which is set whenever a page is accessed. (ARMv8 calls it an *access bit*.) The operating system periodically clears the reference bits and later records them so it can determine which pages were touched during a particular time period. With this usage information, the operating system can select a page that is among the least recently referenced (detected by having its reference bit off). If this bit is not provided by the hardware, the operating system must find another way to estimate which pages have been accessed.

Reference bit: Also called ***use bit*** or ***access bit***. A field that is set whenever a page is accessed and that is used to implement LRU or other replacement schemes.

Figure 5.7.5: ARMv8 uses four levels of tables to translate a 48-bit virtual address into a 40-bit physical address. (COD Figure 5.29).

Rather than needing 64 billion page table entries for the single page table in COD Figure 5.27 (The page table is indexed with the virtual page number ...), this hierarchical approach needs just a tiny fraction. Each step of the translation uses 9 bits of the virtual address to find the next level table, until the upper 36 bits of the virtual address are mapped to the physical address of the desired 4 KiB page. Each ARMv8 page table entry is 8 bytes, so the 512 entries of a table fill a single 4 KiB page. The *Translation Table Base Register* (TTBR) gives the starting address of the first page table.



Virtual memory for large virtual addresses

The caption in COD Figure 5.27 points out that with a single level page table for a 48-bit address with 4 KiB pages, we need 64 billion table entries. As each page table entry is 8 bytes for ARMv8, it would require 0.5 TiB just to map the virtual addresses to physical addresses! Moreover, there could be hundreds of processes running, each with its own page table. That much memory for translation would be unaffordable even for the largest systems.

A range of techniques is used to reduce the amount of storage required for the page table. The five techniques below aim at reducing the total maximum storage required as well as minimizing the main memory dedicated to page tables:

1. The simplest technique is to keep a limit register that restricts the size of the page table for a given process. If the virtual page number becomes larger than the contents of the limit register, entries must be added to the page table. This technique allows the page table to grow as a process consumes more space. Thus, the page table will only be large if the process is using many pages of virtual address space. This technique requires that the address space expand in just one direction.
2. Allowing growth in only one direction is not sufficient, since most languages require two areas whose size is expandable: one area holds the stack and the other area holds the heap. Because of this duality, it is convenient to divide the page table and let it grow from the highest address down, as well as from the lowest address up. This means that there will be two separate page tables and two separate limits. The use of two page tables breaks the address space into two segments. The high-order bit of an address usually determines which segment and thus which page table to use for that address. Since the high-order address bit specifies the segment, each segment can be as large as one-half of the address space. A limit register for each segment specifies the current size of the segment, which grows in units of pages. This type of segmentation is used by many architectures, including ARMv8 and MIPS. Unlike the type of segmentation discussed in a previous elaboration, this form of segmentation is invisible to the application program, although not to the operating system. The major disadvantage of this scheme is that it does not work well when the address space is used in a sparse fashion rather than as a contiguous set of virtual addresses.
3. Another approach to reducing the page table size is to apply a hashing function to the virtual address so that the page table need be only the size of the number of *physical* pages in main memory. Such a structure is called an *inverted page table*. Of course, the lookup process is slightly more complex with an inverted page table, because we can no longer just index the page table.
4. To reduce the actual main memory tied up in page tables, most modern systems also allow the page tables to be paged. Although this sounds tricky, it works by using the same basic ideas of virtual memory and simply allowing the page tables to reside in the virtual address space. In addition, there are some small but critical problems, such as a never-ending series of page faults, which must be avoided. How these problems are overcome is both very detailed and typically highly processor-specific. In brief, these problems are avoided by placing all the page tables in the address space of the operating system and placing at least some of the page tables for the operating system in a portion of main memory that is physically addressed and is always present and thus never in secondary memory.
5. Multiple levels of page tables can also be used to reduce the total amount of page table storage, and this is the solution that ARMv8 uses to reduce the memory footprint of address translation. The figure above shows the four levels of address translation to go from a 48-bit virtual address to a 40-bit physical address of a 4 KiB page. Address translation happens by first looking in the level 0 table, using the highest-order bits of the address. If the address in this table is valid, the next set of high-order bits is used to index the page table indicated by the segment table entry, and so on. Thus, the level 0 table maps the virtual address to a 512 GB (2^{39} bytes) region. The level 1 table in turn maps the virtual address to a 1 GB (2^{30}) region. The next level maps this down to a 2 MB (2^{21}) region. The final table maps the virtual address to the 4 KiB (2^{12}) memory page. This scheme allows the address space to be used in a sparse fashion (multiple noncontiguous segments can be active) without having to allocate the entire page table. Such schemes are particularly useful with very large address spaces and in software systems that require noncontiguous allocation. The primary disadvantage of this multi-level mapping is the more complex process for address translation.

Elaboration

An earlier elaboration mentioned that ARMv8 offers three choices for the minimum page size: 4, 16, and 64 Kibibyte. The caption of the figure above notes that the page table at each level is exactly one page, translating 9 bits of the virtual address per level. If a system uses the larger minimum page sizes, it can map more bits of the virtual address in a single page as well as the page itself being bigger, and thus systems with a 64 KiB page require just three levels of page tables.

PARTICIPATION ACTIVITY 5.7.3: Page faults.

1) When a page fault occurs, the operating system must determine where to put the requested page in ____ ____ .

Check Show answer

2) A ____ address can help to identify the location of a page on disk.

Check Show answer

3) When a process is created, the ____ ____ also creates enough space on disk for all of the process' pages.

Check Show answer

4) The operating system uses an ____ scheme to replace pages in memory when all of the pages are in use?

Check Show answer

What about writes?

The difference between the access time to the cache and main memory is tens to hundreds of cycles, and write-through schemes can be used, although we need a write buffer to hide the latency of the write from the processor. In a virtual memory system, writes to the next level of the hierarchy (disk) can take millions of processor clock cycles; therefore, building a write buffer to allow the system to write-through to disk would be completely impractical. Instead, virtual memory systems must use write-back, performing the individual writes into the page in memory, and copying the page back to secondary memory when it is replaced in the main memory.

Hardware/Software Interface

A write-back scheme has another major advantage in a virtual memory system. Because the disk transfer time is small compared with its access time, copying back an entire page is much more efficient than writing individual words back to the disk. A write-back operation, although faster than transferring separate words, is still costly. Thus, we would like to know whether a page *needs* to be copied back when we choose to replace it. To track whether a page has been written since it was read into the memory, a *dirty bit* is added to the page table. The dirty bit is set when any word in a page is written. If the operating system chooses to replace the page, the dirty bit indicates whether the page needs to be written out before its location in memory can be given to another page. Hence, a modified page is often called a *dirty* page.

PARTICIPATION ACTIVITY 5.7.4: Writes.

1) In a virtual memory system, an ____ is typically written to the disk.
   - ○ individual word
   - ○ entire page

2) A ____ bit indicates if a page has been written since being read into memory.
   - ○ dirty
   - ○ use

Making address translation fast: The TLB

Since the page tables are stored in main memory, every memory access by a program can take at least twice as long: one memory access to obtain the physical address and a second access to get the data. The key to improving access performance is to rely on locality of reference to the page table. When a translation for a virtual page number is used, it will probably be needed again soon, because the references to the words on that page have both temporal and spatial locality.

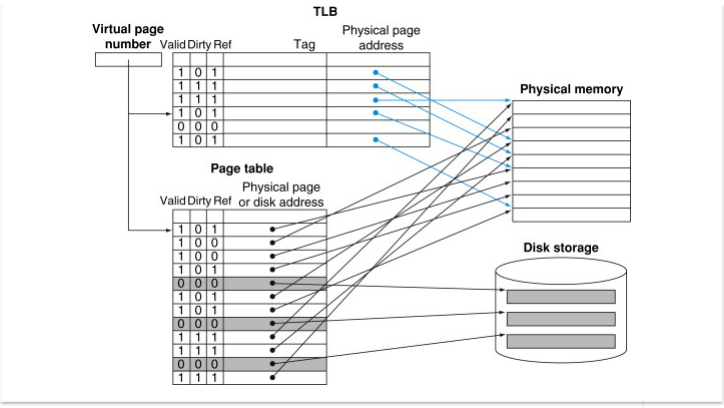
Accordingly, modern processors include a special cache that keeps track of recently used translations. This special address translation cache is traditionally referred to as a *translation-lookaside buffer* (TLB), although it would be more accurate to call it a translation cache. The TLB corresponds to that little piece of paper we typically use to record the location of a set of books we look up in the card catalog; rather than continually searching the entire catalog, we record the location of several books and use the scrap of paper as a cache of Library of Congress call numbers.

Translation-lookaside buffer (TLB): A cache that keeps track of recently used address mappings to try to avoid an access to the page table.

The figure below shows that each tag entry in the TLB holds a portion of the virtual page number, and each data entry of the TLB holds a physical page number. Because we access the TLB instead of the page table on every reference, the TLB will need to include other status bits, such as the dirty and the reference bits. Although the figure below shows a single page table, TLBs work fine with multi-level page tables as well. The TLB simply loads the physical address and protection tags from the last level page table.

Figure 5.7.6: The TLB acts as a cache of the page table for the entries that map to physical pages only (COD Figure 5.30).

The TLB contains a subset of the virtual-to-physical page mappings that are in the page table. The TLB mappings are shown in color. Because the TLB is a cache, it must have a tag field. If there is no matching entry in the TLB for a page, the page table must be examined. The page table either supplies a physical page number for the page (which can then be used to build a TLB entry) or indicates that the page resides on disk, in which case a page fault occurs. Since the page table has an entry for every virtual page, no tag field is needed; in other words, unlike a TLB, a page table is *not* a cache.



On every reference, we look up the virtual page number in the TLB. If we get a hit, the physical page number is used to form the address, and the corresponding reference bit is turned on. If the processor is performing a write, the dirty bit is also turned on. If a miss in the TLB occurs, we must determine whether it is a page fault or merely a TLB miss. If the page exists in memory, then the TLB miss indicates only that the translation is missing. In such cases, the processor can handle the TLB miss by loading the translation from the (last-level) page table into the TLB and then trying the reference again. If the page is not present in memory, then the TLB miss indicates a true page fault. In this case, the processor invokes the operating system using an exception. Because the TLB has many fewer entries than the number of pages in main memory, TLB misses will be much more frequent than true page faults.

TLB misses can be handled either in hardware or in software. In practice, with care there can be little performance difference between the two approaches, because the basic operations are the same in either case.

After a TLB miss occurs and the missing translation has been retrieved from the page table, we will need to select a TLB entry to replace. Because the reference and dirty bits are contained in the TLB entry, we need to copy these bits back to the page table entry when we replace an entry. These bits are the only portion of the TLB entry that can be changed. Using write-back—that is, copying these entries back at miss time rather than when they are written—is very efficient, since we expect the TLB miss rate to be small. Some systems use other techniques to approximate the reference and dirty bits, eliminating the need to write into the TLB except to load a new table entry on a miss.

Some typical values for a TLB might be

- TLB size: 16-512 entries
- Block size: 1-2 page table entries (typically 4-8 bytes each)
- Hit time: 0.5-1 clock cycle
- Miss penalty: 10-100 clock cycles
- Miss rate: 0.01%-1%

Designers have used a wide variety of associativities in TLBs. Some systems use small, fully associative TLBs because a fully associative mapping has a lower miss rate; furthermore, since the TLB is small, the cost of a fully associative mapping is not too high. Other systems use large TLBs, often with small associativity. With a fully associative mapping, choosing the entry to replace becomes tricky since implementing a hardware LRU scheme is too expensive. Furthermore, since TLB misses are much more frequent than page faults and thus must be handled more cheaply, we cannot afford an expensive software algorithm, as we can for page faults. As a result, many systems provide some support for randomly choosing an entry to replace. We'll examine replacement schemes in a little more detail in COD Section 5.8 (A common framework for memory hierarchy).

PARTICIPATION ACTIVITY 5.7.5: The TLB.

write-back status bits fewer cache

The TLB, or translation-lookaside buffer, is a special _____ that keeps track of recently used translations.

Each TLB entry includes the physical page address, tag, and _____ .

A TLB has _____ number of entries as a page table.

A TLB entry is replaced using a _____ scheme.

Reset

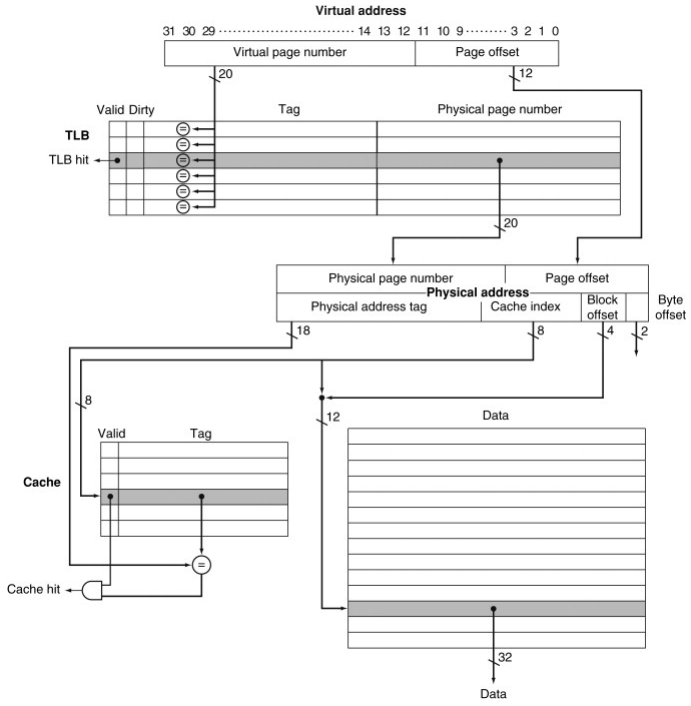
The Intrinsity FastMATH TLB

To see these ideas in a real processor, let's take a closer look at the TLB of the Intrinsity FastMATH. The memory system uses 4 KiB pages and just a 32-bit address space; thus, the virtual page number is 20 bits long. The physical address is the same size as the virtual address. The TLB contains 16 entries, it is fully associative, and it is shared between the instruction and data references. Each entry is 64 bits wide and contains a 20-bit tag (which is the virtual page number for that TLB entry), the corresponding physical page number (also 20 bits), a valid bit, a dirty bit, and other bookkeeping bits. Like most ARMv8 systems, it uses software to handle TLB misses.

Figure 5.7.7: The TLB and cache implement the process of going from a virtual address to a data item in the Intrinsity FastMATH (COD Figure 5.31).

This figure shows the organization of the TLB and the data cache, assuming a 4 KiB page size. Note that the address size for this computer is just 32 bits. This diagram focuses on a read; the figure below describes how to handle writes. Note that unlike COD Figure 5.12 (The 16 KiB caches in the Intrinsity FastMATH ...), the tag and data RAMs are split. By addressing the long but narrow data

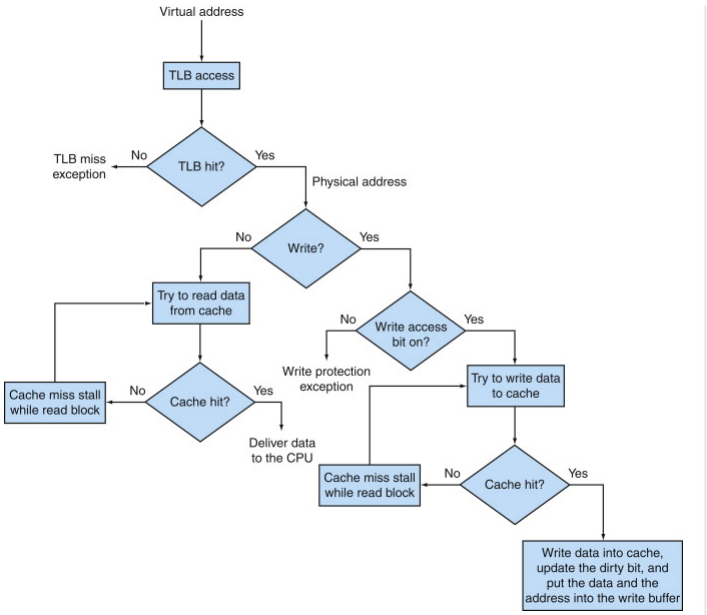
RAM with the cache index concatenated with the block offset, we select the desired word in the block without a 16:1 multiplexor. While the cache is direct mapped, the TLB is fully associative. Implementing a fully associative TLB requires that every TLB tag be compared against the virtual page number, since the entry of interest can be anywhere in the TLB. If the valid bit of the matching entry is on, the access is a TLB hit, and bits from the physical page number together with bits from the page offset form the index that is used to access the cache.



The figure above shows the TLB and one of the caches, while the figure below shows the steps in processing a read or write request. When a TLB miss occurs, the hardware saves the page number of the reference in a special register and generates an exception. The exception invokes the operating system, which handles the miss in software. To find the physical address for the missing page, a TLB miss indexes the page table using the page number of the virtual address and the page table register, which indicates the starting address of the active process page table. Using a special set of system instructions that can update the TLB, the operating system places the physical address from the page table into the TLB. A TLB miss takes about 13 clock cycles, assuming the code and the page table entry are in the instruction cache and data cache, respectively. A true page fault occurs if the page table entry does not have a valid physical address. The hardware maintains an index that indicates the recommended entry to replace; it is chosen randomly.

Figure 5.7.8: Processing a read or a write-through in the Intrinsity FastMATH TLB and cache (COD Figure 5.32).

If the TLB generates a hit, the cache can be accessed with the resulting physical address. For a read, the cache generates a hit or miss and supplies the data or causes a stall while the data are brought from memory. If the operation is a write, a portion of the cache entry is overwritten for a hit and the data are sent to the write buffer if we assume write-through. A write miss is just like a read miss except that the block is modified after it is read from memory. Write-back requires writes to set a dirty bit for the cache block, and a write buffer is loaded with the whole block only on a read miss or write miss if the block to be replaced is dirty. Notice that a TLB hit and a cache hit are independent events, but a cache hit can only occur after a TLB hit occurs, which means that the data must be present in memory. The relationship between TLB misses and cache misses is examined further in the following example and the exercises at the end of this chapter. Note that the address size for this computer is just 32 bits.



There is an extra complication for write requests: namely, the write access bit in the TLB must be checked. This bit prevents the program from writing into pages for which it has only read access. If the program attempts a write and the write access bit is off, an exception is generated. The write access bit forms part of the protection mechanism, which we will discuss shortly.

PARTICIPATION ACTIVITY 5.7.6: The Intrinsity FastMATH TLB.

1) Most ARMv8 systems use hardware to handle TLB misses.
   - ○ True
   - ○ False

2) In the FastMATH TLB, an entry's TLB tag is also the entry's virtual page number.
   - ○ True
   - ○ False

3) A TLB miss indicates ____ .
   - ○ a page fault
   - ○ a referenced page is not in the TLB

4) Which of the following occurs if the Intrinsity FastMATH system attempts to write to a page who's write bit is off?
   - ○ Cache miss stall for read
   - ○ Write protection exception
   - ○ Cache miss stall for write

Integrating virtual memory, TLBs, and caches

Our virtual memory and cache systems work together as a hierarchy, so that data cannot be in the cache unless it is present in main memory. The operating system helps maintain this hierarchy by flushing the contents of any page from the cache when it decides to migrate that page to secondary memory. At the same time, the OS modifies the page tables and TLB, so that an attempt to access any data on the migrated page will generate a page fault.

Under the best of circumstances, a virtual address is translated by the TLB and sent to the cache where the appropriate data are found, retrieved, and sent back to the processor. In the worst case, a reference can miss in all three components of the memory hierarchy: the TLB, the page table, and the cache. The following example illustrates these interactions in more detail.

Example 5.7.1: Overall operation of a memory hierarchy.

In a memory hierarchy like that of COD Figure 5.31 (The TLB and cache …), which includes a TLB and a cache organized as shown, a memory reference can encounter three different types of misses: a TLB miss, a page fault, and a cache miss. Consider all the combinations of these three events with one or more occurring (seven possibilities). For each possibility, state whether this event can actually occur and under what circumstances.

Answer

The figure below shows all combinations and whether each is possible in practice.

Figure 5.7.9: The possible combinations of events in the TLB, virtual memory system, and cache (COD Figure 5.33).

Three of these combinations are impossible, and one is possible (TLB hit, page table hit, cache miss) but never detected.

TLB	Page table	Cache	Possible? If so, under what circumstance?
Hit	Hit	Miss	Possible, although the page table is never really checked if TLB hits.
Miss	Hit	Hit	TLB misses, but entry found in page table; after retry, data is found in cache.
Miss	Hit	Miss	TLB misses, but entry found in page table; after retry, data misses in cache.
Miss	Miss	Miss	TLB misses and is followed by a page fault; after retry, data must miss in cache.
Hit	Miss	Miss	Impossible: cannot have a translation in TLB if page is not present in memory.
Hit	Miss	Hit	Impossible: cannot have a translation in TLB if page is not present in memory.
Miss	Miss	Hit	Impossible: data cannot be allowed in cache if the page is not in memory.

PARTICIPATION ACTIVITY 5.7.7: Virtual memory miss combinations.

Are the following TLB, virtual memory system, and cache miss combinations possible or impossible?

1) TLB hit, page table miss, cache miss
   - ○ Possible
   - ○ Impossible

2) TLB miss, page table miss, cache hit
   - ○ Possible
   - ○ Impossible

3) TLB hit, page table hit, cache miss
   - ○ Possible
   - ○ Impossible

4) TLB hit, page table miss, cache hit
   - ○ Possible
   - ○ Impossible

Elaboration

The figure above assumes that all memory addresses are translated to physical addresses before the cache is accessed. In this organization, the cache is physically indexed and physically tagged (both the cache index and tag are physical, rather than virtual, addresses). In such a system, the amount of time to access memory, assuming a cache hit, must accommodate both a TLB access and a cache access; of course, these accesses can be ***pipelined****.*

Alternatively, the processor can index the cache with an address that is completely or partially virtual. This is called a virtually addressed cache, and it uses tags that are virtual addresses; hence, such a cache is virtually indexed and virtually tagged. In such caches, the address translation hardware (TLB) is unused during the normal cache access, since the cache is accessed with a virtual address that has not been translated to a physical address. This takes the TLB out of the critical path, reducing cache latency. When a cache miss occurs, however, the processor needs to translate the address to a physical address so that it can fetch the cache block from main memory.

When the cache is accessed with a virtual address and pages are shared between processes (which may access them with different virtual addresses), there is the possibility of aliasing. Aliasing occurs when the same object has two names—in this case, two virtual addresses for the same page. This ambiguity creates a problem, because a word on such a page may be cached in two different locations, each corresponding to distinct virtual addresses. This ambiguity would allow one program to write the data without the other program being aware that the data had changed. Completely virtually addressed caches either introduce design limitations on the cache and TLB to reduce aliases or require the operating system, and possibly the user, to take steps to ensure that aliases do not occur.

A common compromise between these two design points is caches that are virtually indexed—sometimes using just the page-offset portion of the address, which is really a physical address since it is not translated—but use physical tags. These designs, which are virtually indexed but physically tagged, attempt to achieve the performance advantages of virtually indexed caches with the architecturally simpler advantages of a physically addressed cache. For example, there is no alias problem in this case. COD Figure 5.31 (The TLB and cache …) assumed a 4 KiB page size, but it's really 16 KiB, so the Intrinsity FastMATH can use this trick. To pull it off, there must be careful coordination between the minimum page size, the cache size, and associativity. ARMv8 allows instruction caches to use either physical or virtual indexing, but they must be physically tagged. It requires data caches to behave as though physically tagged and indexed, but it does not mandate this implementation. For example, virtually indexed, physically tagged data caches could use additional logic to ensure that their behavior is consistent with the ARMv8 definition.

Virtually addressed cache: A cache that is accessed with a virtual address rather than a physical address.

Aliasing: A situation in which two addresses access the same object; it can occur in virtual memory when there are two virtual addresses for the same physical page.

Physically addressed cache: A cache that is addressed by a physical address.

PARTICIPATION ACTIVITY 5.7.8: Virtual caches.

Aliasing | Virtually addressed cache | Physically addressed cache

Occurs when two virtual addresses access the same page.

A cache that is accessed by a physical address.

A cache that is accessed by a partially or completely virtual address.

Reset

Implementing protection with virtual memory

Perhaps the most important function of virtual memory today is to allow sharing of a single main memory by multiple processes, while providing memory protection among these processes and the operating system. The protection mechanism must ensure that although multiple processes are sharing the same main memory, one renegade process cannot write into the address space of another user process or into the operating system either intentionally or unintentionally. The write access bit in the TLB can protect a page from being written. Without this level of protection, computer viruses would be even more widespread.

Hardware/Software Interface

To enable the operating system to implement protection in the virtual memory system, the hardware must provide at least the three basic capabilities summarized below. Note that the first two are the same requirements as needed for virtual machines (COD Section 5.6 (Virtual machines)).

1. Support at least two modes that indicate whether the running process is a user process or an operating system process, variously called a *supervisor* process, a *kernel* process, or an *executive* process.
2. Provide a portion of the processor state that a user process can read but not write. This state includes the user/supervisor mode bit, which dictates whether the processor is in user or supervisor mode, the page table pointer, and the TLB. To write these elements, the operating system uses special instructions that are only available in supervisor mode.
3. Provide mechanisms whereby the processor can go from user mode to supervisor mode and vice versa. The first direction is typically accomplished by a *system call* exception, implemented as a special instruction (*SVC* in the ARMv8 instruction set) that transfers control to a dedicated location in supervisor code space. As with any other exception, the program counter from the point of the system call is saved in the *exception link register* (ELR), and the processor is placed in supervisor mode. To return to user mode from the exception, use the *exception return* (ERET) instruction, which resets to user mode and jumps to the address in ELR.

By using these mechanisms and storing the page tables in the operating system's address space, the operating system can change the page tables while preventing a user process from changing them, ensuring that a user process can access only the storage provided to it by the operating system.

Supervisor mode: Also called ***kernel mode***. A mode indicating that a running process is an operating system process.

System call: A special instruction that transfers control from user mode to a dedicated location in supervisor code space, invoking the exception mechanism in the process.

We also want to prevent a process from reading the data of another process. For example, we wouldn't want a student program to read the teacher's grades while they were in the processor's memory. Once we begin sharing main memory, we must provide the ability for a process to protect its data from both reading and writing by another process; otherwise, sharing the main memory will be a mixed blessing!

Remember that each process has its own virtual address space. Thus, if the operating system keeps the page tables organized so that the independent virtual pages map to disjoint physical pages, one process will not be able to access another's data. Of course, this also requires that a user process be unable to change the page table mapping. The operating system can assure safety if it prevents the user process from modifying its own page tables. However, the operating system must be able to modify the page tables. Placing the page tables in the protected address space of the operating system satisfies both requirements.

When processes want to share information in a limited way, the operating system must assist them, since accessing the information of another process requires changing the page table of the accessing process. The write access bit can be used to restrict the sharing to just read sharing, and, like the rest of the page table, this bit can be changed only by the operating system. To allow another process, say, P1, to read a page owned by process P2, P2 would ask the operating system to create a page table entry for a virtual page in P1's address space that points to the same physical page that P2 wants to share. The operating system could use the write protection bit to prevent P1 from writing the data, if that was P2's wish. Any bits that determine the access rights for a page must be included in both the page table and the TLB, because the page table is accessed only on a TLB *miss*.

Elaboration

When the operating system decides to change from running process P1 to running process P2 (called a context switch or process switch), it must ensure that P2 cannot get access to the page tables of P1 because that would compromise protection. If there is no TLB, it suffices to change the page table register to point to P2's page table (rather than to P1's); with a TLB, we must clear the TLB entries that belong to P1—both to protect the data of P1 and to force the TLB to load the entries for P2. If the process switch rate were high, this could be quite inefficient. For example, P2 might load only a few TLB entries before the operating system switched back to P1. Unfortunately, P1 would then find that all its TLB entries were gone and would have to pay TLB misses to reload them. This problem arises because the virtual addresses used by P1 and P2 are the same, and we must clear out the TLB to avoid confusing these addresses.

A common alternative is to extend the virtual address space by adding a process identifier or task identifier. The Intrinsity FastMATH has an 8-bit address space ID (ASID) field for this purpose. This small field identifies the currently running process; it is kept in a register loaded by the operating system when it switches processes. ARMv8 also offers ASID to reduce TLB flushes on context switches. The process identifier is concatenated to the tag portion of the TLB, so that a TLB hit occurs only if both the page number and the process identifier match. This combination eliminates the need to clear the TLB, except on rare occasions.

Similar problems can occur for a cache, since on a process switch, the cache will contain data from the running process. These problems arise in different ways for physically addressed and virtually addressed caches, and a variety of solutions, such as process identifiers, are used to ensure that a process gets its own data.

Context switch: A changing of the internal state of the processor to allow a different process to use the processor that includes saving the state needed to return to the currently executing process.

PARTICIPATION ACTIVITY 5.7.9: Virtual memory protection.

1) Without the write access bit, computer viruses would likely be more widespread.
   - ○ True
   - ○ False

2) For virtual memory system protection, hardware does not need to determine if running processes are user or supervisor processes.
   - ○ True
   - ○ False

3) syscall is a special instruction that enables a processor to go from user mode to supervisor mode.
   - ○ True
   - ○ False

Handling TLB misses and page faults

Although the translation of virtual to physical addresses with a TLB is straightforward when we get a TLB hit, as we saw earlier, handling TLB misses and page faults is more complex. A TLB miss occurs when no entry in the TLB matches a virtual address. Recall that a TLB miss can indicate one of two possibilities:

1. The page is present in memory, and we need only create the missing TLB entry.
2. The page is not present in memory, and we need to transfer control to the operating system to deal with a page fault.

Handling a TLB miss or a page fault requires using the exception mechanism to interrupt the active process, transferring control to the operating system, and later resuming execution of the interrupted process. A page fault will be recognized sometime during the clock cycle used to access memory. To restart the instruction after the page fault is handled, the program counter of the instruction that caused the page fault must be saved. The *exception link register* (ELR) is used to hold this value.

In addition, a TLB miss or page fault exception must be asserted by the end of the same clock cycle that the memory access occurs, so that the next clock cycle will begin exception processing rather than continue normal instruction execution. If the page fault was not recognized in this clock cycle, a load instruction could overwrite a register, and this could be disastrous when we try to restart the instruction. For example, consider the instruction `LDUR X1, [X1,#0]`: the computer must be able to prevent the write pipeline stage from occurring; otherwise, it could not properly restart the instruction, since the contents of `X1` would have been destroyed. A similar complication arises on stores. We must prevent the write into memory from actually completing when there is a page fault; this is usually done by deasserting the write control line to the memory.

Hardware/Software Interface

Between the time we begin executing the exception handler in the operating system and the time that the operating system has saved all the state of the process, the operating system is particularly vulnerable. For instance, if another exception occurred when we were processing the first exception in the operating system, the control unit would overwrite the exception link register, making it impossible to return to the instruction that caused the page fault! We can avoid this disaster by providing the ability to *disable* and *enable exceptions*. When an exception first occurs, the processor sets a bit that disables all other exceptions; this could happen at the same time the processor sets the supervisor mode bit. The operating system

will then save just enough state to allow it to recover if another exception occurs—namely, the *exception link register* (ELR) and the *exception syndrome register* (ESR), which as we saw in COD Chapter 4 (The Processor) records the reason for the exception. ELR and ESR in ARMv8 are two of the special control registers that help with exceptions, TLB misses, and page faults. The operating system can then re-enable exceptions. These steps make sure that exceptions will not cause the processor to lose any state and thereby be unable to restart execution of the interrupting instruction.

Exception enable: Also called interrupt enable. A signal or action that controls whether the process responds to an exception or not; necessary for preventing the occurrence of exceptions during intervals before the processor has safely saved the state needed to restart.

Once the operating system knows the virtual address that caused the page fault, it must complete three steps:

1. Look up the page table entry using the virtual address and find the location of the referenced page in secondary memory.
2. Choose a physical page to replace; if the chosen page is dirty, it must be written out to secondary memory before we can bring a new virtual page into this physical page.
3. Start a read to bring the referenced page from secondary memory into the chosen physical page.

Of course, this last step will take millions of processor clock cycles for disks (so will the second if the replaced page is dirty); accordingly, the operating system will usually select another process to execute in the processor until the disk access completes. Because the operating system has saved the state of the process, it can freely give control of the processor to another process.

When the read of the page from secondary memory is complete, the operating system can restore the state of the process that originally caused the page fault and execute the instruction that returns from the exception. This instruction will reset the processor from kernel to user mode, as well as restore the program counter. The user process then re-executes the instruction that faulted, accesses the requested page successfully, and continues execution.

Page fault exceptions for data accesses are difficult to implement properly in a processor because of a combination of three characteristics:

1. They occur in the middle of instructions, unlike instruction page faults.
2. The instruction cannot be completed before handling the exception.
3. After handling the exception, the instruction must be restarted as if nothing had occurred.

Making instructions *restartable*, so that the exception can be handled and the instruction later continued, is relatively easy in an architecture like the ARMv8. Because each instruction writes only one data item and this write occurs at the end of the instruction cycle, we can simply prevent the instruction from completing (by not writing) and restart the instruction at the beginning.

Restartable instruction: An instruction that can resume execution after an exception is resolved without the exception's affecting the result of the instruction.

PARTICIPATION ACTIVITY 5.7.10: Handling TLB misses and page faults.

1) A TLB miss or page fault _____ must be asserted by the end of the same clock cycle as the memory access.

Check Show answer

2) The exception _____ signal controls whether a process responds to an exception.

Check Show answer

3) The _____ and ESR are two ARMv8 control registers that help with page faults, TLB misses, and exceptions.

Check Show answer

4) A _____ instruction can resume or restart execution after being interrupted by an exception without affecting the result of the instruction.

Check Show answer

Elaboration

For processors with more complex instructions that can touch many memory locations and write many data items, making instructions restartable is much harder. Processing one instruction may generate a number of page faults in the middle of the instruction. For example, x86 processors have block move instructions that touch thousands of data words. In such processors, instructions often cannot be restarted from the beginning, as we do for ARMv8 instructions. Instead, the instruction must be interrupted and later continued midstream in its execution. Resuming an instruction in the middle of its execution usually requires saving some special state, processing the exception, and restoring that special state. Making this work

properly requires careful and detailed coordination between the exception-handling code in the operating system and the hardware.

Elaboration

Rather than pay an extra level of indirection on every memory access, the Virtual Memory Monitor (COD Section 5.6 (Virtual machines)) maintains a shadow page table that maps directly from the guest virtual address space to the physical address space of the hardware. By detecting all modifications to the guest's page table, the VMM can ensure the shadow page table entries being used by the hardware for translations correspond to those of the guest OS environment, with the exception of the correct physical pages substituted for the real pages in the guest tables. Hence, the VMM must trap any attempt by the guest OS to change its page table or to access the page table pointer. This is commonly done by write protecting the guest page tables and trapping any access to the page table pointer by a guest OS. As noted above, the latter happens naturally if accessing the page table pointer is a privileged operation.

Elaboration

The final portion of the architecture to virtualize is I/O. This is by far the most difficult part of system virtualization because of the increasing number of I/O devices attached to the computer and the expanding diversity of I/O device types. Another difficulty is the sharing of a real device among multiple VMs, and yet another comes from supporting the myriad of device drivers that are required, especially if different guest OSes are supported on the same VM system. The VM illusion can be maintained by giving each VM generic versions of each type of I/O device driver, and then leaving it to the VMM to handle real I/O.

Elaboration

In addition to virtualizing the instruction set for a virtual machine, another challenge is virtualization of virtual memory, as each guest OS in every virtual machine manages its own set of page tables. To make this work, the VMM separates the notions of real and physical memory (which are often treated synonymously), and makes real memory a separate, intermediate level between virtual memory and physical memory. (Some use the terms virtual memory, physical memory, and machine memory to name the same three levels.) The guest OS maps virtual memory to real memory via its page tables, and the VMM page tables map the guest's real memory to physical memory. The virtual memory architecture is typically specified via page tables, as in IBM VM/370, the x86, and ARMv8.

Summary

Virtual memory is the name for the level of memory hierarchy that manages caching between the main memory and secondary memory. Virtual memory allows a single program to expand its address space beyond the limits of main memory. More importantly, virtual memory supports sharing of the main memory among multiple, simultaneously active processes, in a protected manner.

Managing the memory hierarchy between main memory and disk is challenging because of the high cost of page faults. Several techniques are used to reduce the miss rate:

1. Pages are made large to take advantage of spatial locality and to reduce the miss rate.
2. The mapping between between virtual addresses and physical addresses, which is implemented with a page table, is made fully associative so that a virtual page can be placed anywhere in main memory.
3. The operating system uses techniques, such as LRU and a reference bit, to choose which pages to replace.

Writes to secondary memory are expensive, so virtual memory uses a write-back scheme and also tracks whether a page is unchanged (using a dirty bit) to avoid writing clean pages.

The virtual memory mechanism provides address translation from a virtual address used by the program to the physical address space used for accessing memory. This address translation allows protected sharing of the main memory and provides several additional benefits, such as simplifying memory allocation. Ensuring that processes are protected from each other requires that only the operating system can change the address translations, which is implemented by preventing user programs from altering the page tables. Controlled sharing of pages between processes can be implemented with the help of the operating system and access bits in the page table that indicate whether the user program has read or write access to a page.

If a processor had to access a page table resident in memory to translate every access, virtual memory would be too expensive, as caches would be pointless! Instead, a TLB acts as a cache for translations from the page table. Addresses are then translated from virtual to physical using the translations in the TLB.

Caches, virtual memory, and TLBs all rely on a common set of principles and policies. The next section discusses this common framework.

Understanding program performance

Although virtual memory was invented to enable a small memory to act as a large one, the performance difference between secondary memory and main memory means that if a program routinely accesses more virtual memory than it has physical memory, it will run very slowly. Such a program would be continuously swapping pages between main memory and secondary memory, called *thrashing*. Thrashing is a disaster if it occurs, but it is rare. If your program thrashes, the easiest solution is to run it on a computer with more memory or buy more memory for your computer. A more complex choice is to re-examine your algorithm and data structures to see if you can change the locality and thereby reduce the number of pages that your program uses simultaneously. This set of popular pages is informally called the *working set*.

A more common performance problem is TLB misses. Since a TLB might handle only 32-64 page entries at a time, a program could easily see a high TLB miss rate, as the processor may access less than a quarter mebibyte directly: 64 × 4 KiB = 0.25 MiB. For example, TLB misses are often a challenge for Radix Sort. To try to alleviate this problem, most computer

architectures now offer support for larger page sizes. For instance, in addition to the minimum 4 KiB page, ARMv8 hardware supports 2 MiB and 1 GiB pages. Hence, if a program uses large page sizes, it can access more memory directly without TLB misses.

The practical challenge is getting the operating system to allow programs to select these larger page sizes. Once again, the more complex solution to reducing TLB misses is to re-examine the algorithm and data structures to reduce the working set of pages; given the importance of memory accesses to performance and the frequency of TLB misses, some programs with large working sets have been redesigned with that goal.

Elaboration

ARMv8 supports the larger page sizes via the multi-level page table of COD Figure 5.29 (ARMv8 uses four levels of tables …). In addition to pointing at the next level page table in levels 1 and 2, it allows a block translation to map the virtual address to a 1 GiB physical address (if the block translation is in level 1) or a 2 MiB physical address (if the block translation is in level 2). If the minimum page size is larger than 4 KiB, the block translations are also larger: 64 GiB and 32 MiB for 16 KiB pages and 4096 GiB and 512 MiB for 64 KiB pages.

Elaboration

ARMv8 has many options and optimizations that we do not have space to cover; the description of virtual memory takes 200 pages in the ARMv8 architecture manual. For example, while the minimum system has just two exception levels (EL0 and EL1), to support virtual machines monitors there is an optional third exception level (EL2) and a fourth level (EL3) for security monitors. There are versions of the special ELR and ESR registers for each level. To get greater performance from the TLB, ARMv8 offers a hint that a single entry corresponds to 16 contiguous ranges that have the same permissions and attributes, thereby expanding the reach of that entry by a factor of 16. To offer different types of shared addresses for cores within a chip versus a cluster of chips, ARMv8 distinguishes inner sharability versus outer sharability, with the former always sharing the same domain as the latter.

PARTICIPATION ACTIVITY 5.7.11: Check yourself.

L1 cache | Main memory | TLB | L2 cache

A cache for a cache.

A cache for main memory.

A cache for disks.

A cache for page table entries.

Reset

Provide feedback on this section