Introduction to Operating Systems

Virtual Memory

Contents

List of Figures

List of Tables

1 Acknowledgments

As with any Computer Science course, Introduction to Operating Systems is constantly evolving. This evolution does not happen in a vacuum. In addition to many outside resources being consulted, we have been very fortunate to have important contributions from our TAs. Ted Cooper and Jeremiah Peschka have been particularly valuable in improving these lecture notes, as editors and contributors. Ted and Matt Spohrer developed the quiz infrastructure and questions. The quiz questions form the basis of the study questions now included with each set of notes. Ted and Jeremiah have made significant contributions and improvements to the course projects which use the xv6 teaching operating system.

As always, any omissions and/or errors are mine, notifications of such are appreciated.

2 Learning Objectives

- Discuss the principle objectives and requirements for memory management.
- Explain the strengths and weaknesses of directly using physical memory and not use virtual memory.
- Explain the concept of *paging*.
- Explain the concept of *segmentation*.
- Explain certain memory management schemes.
- Explain certain techniques for implementing segmented memory.
- Explain under what conditions segmentation is superior to paging and vice-versa.
- Explain how `malloc()` and similar memory management routines operate.
- Describe hardware control structures used to support virtual memory.
- Explain the impact of virtual memory on the programming model for user programs.
- Explain the memory management techniques used in Unix and Linux.
- Define and explain

  ▷ Virtual Memory
  ▷ Isolation Principle
  ▷ Memory Management Unit
  ▷ Locality of Reference
  ▷ Working Set
  ▷ Relocation
  ▷ Translation Lookaside Buffer
  ▷ Page table, single- and multi-level
  ▷ Virtual Page
  ▷ Physical Page Frame
  ▷ Free space management
  ▷ Page replacement algorithms: FIFO, LRU, optimal, Modified Clock
  ▷ The strengths and weaknesses of fixed size vs. variable sized segments.
  ▷ Reverse Page Table
  ▷ Buddy Allocator

- Explain the role of the TLB in improving address translation performance for paged VM systems.
- Explain what is meant by *fast path* and *slow path* in paged VM address translation.
- Explain the role of a hard disk in modern paged VM systems.

3 Concepts and Terms

Term	Description
Page Frame	A fixed-length block of main memory. Also called a *physical page*.
Virtual Page	A fixed-length contiguous block of virtual memory, described by a single entry in the page table. It is the smallest unit of data for memory management in a virtual memory operating system. When the term *page* is used without context, it refers to a virtual page.
Segment	A variable-length block of data that resides in secondary memory. An entire segment may temporarily be copied into an available region of main memory (segmentation) or the segment may be divided into pages which can be individually copied into main memory (combined segmentation and paging).
Physical Address	A memory address that is enables access to a particular storage cell of main memory.
Offset	An offset is an integer indicating the distance (displacement) between the beginning of the object and a given location or point within the same object.
Page Offset	The last bits of the virtual address are called the offset which is the difference between the byte address you want and the start of the page. You require enough bits in the offset to access any byte within the page. For a 4K page you require 12 bits of offset ($4K == 2^{12}$). Remember that the smallest amount of memory that the operating system or hardware deals with is a page, so each of these 4096 bytes reside within a single page and are dealt with as a single unit.
Segment Offset	The last bits of the virtual address are called the offset which is the difference between the byte address you want and the start of the segment. You require enough bits in the offset to be able to get to any byte in the segment. For any given computer system, there is a maximum segment size (MSS).
Address Translation	On a system with virtual memory, the addresses used by the processor are *virtual* addresses. However, physical memory can only be accessed with *physical* addresses. The process of converting, or translating, a virtual to physical address is known as address translation. Different mechanisms are used depending on if paging or segmentation is in use.

Term	Description
Virtual Address Space	A virtual address space (VAS), or just address space (AS), is the set of ranges of virtual addresses that an operating system makes available to a process. The range of virtual addresses usually starts at a low address and can extend to the highest address allowed by the computer's instruction set architecture and supported by the operating system's pointer size implementation, which can be 4 bytes for 32-bit or 8 bytes for 64-bit systems. This provides several benefits, one of which is security through *process isolation* assuming each process is given a separate address space.
Base Register	In a segmented memory system, contains the memory address of the *start* of the segment. This is not the same as the lowest address in the segment.
Bounds Register	In a segmented memory system with variable sized segments, contains either: 1) the last valid address in the segment; or 2) the size of the segment.
Working Set	The collection of information referenced by the process during the process time interval $(t-\tau, t)$. Typically the units of information are memory pages. This is an approximation of the set of pages that the process will access in the future (say during the next τ time units), and more specifically is suggested to be an indication of what pages ought to be kept in main memory to allow most progress to be made in the execution of that process.
Internal Fragmentation[1]	Due to the rules governing memory allocation, more computer memory may be allocated than is requested. For example, if memory can only be provided to programs in "chunks" (e.g., multiple of 4), then if a program requests 29 bytes, it will actually get a chunk of 32 bytes. When this happens, the excess memory goes to waste. This waste is called internal fragmentation as it occurs within a region allocated to some process. Unlike other types of fragmentation, internal fragmentation is difficult to reclaim; usually the best way to remove it is with a design change.
External Fragmentation	External fragmentation arises when "chunks" of free memory are separated into small blocks and is interspersed by allocated memory. The result is that, although free storage is available, it is effectively unusable because it is divided into pieces that are too small individually to satisfy the demands of a process. The term "external" refers to the fact that the unusable storage is outside the allocated regions.

Table 1: Memory Management Terms

[1] See Wikipedia.

3.1 Additional Information

1. A seminal paper on the evolution of memory is *Virtual Memory*, Peter J. Denning, Computing Surveys, Vol. 2, No. 3, September 1970.

2. *Virtual Memory: Issues of implementation*, B. Jacob and T. Mudge, in Computer, vol. 31, no. 6, pp. 33-43, June 1998.

3. *Virtual Memory in Contemporary Microprocessors*, Bruce Jacob and Trevor Mudge, IEEE Micro, vol. 18, no. 4, pp. 60-75, July-Aug. 1998.

4 Physical Memory

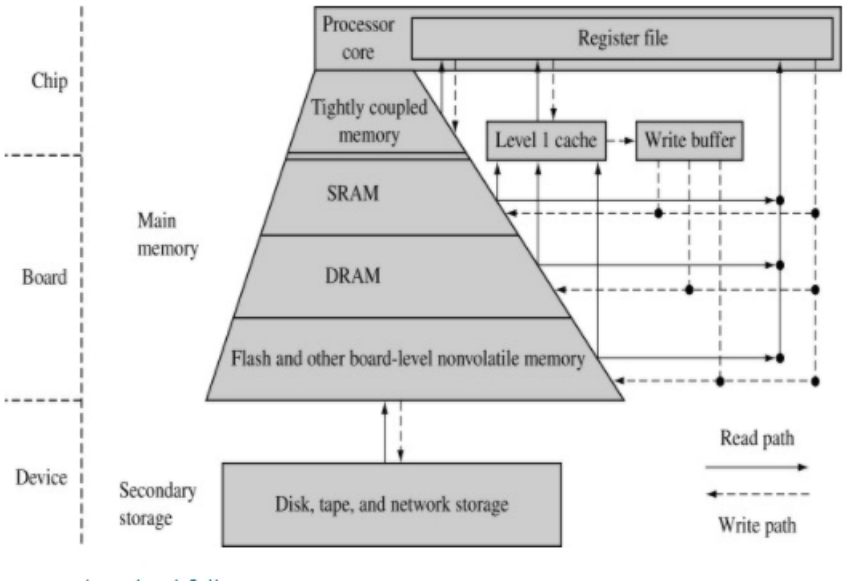


Figure 1: Memory Hierarchy

In modern systems, random-access main memory is typically packaged as a DIMM[2]. These modules are mounted on a printed circuit board and designed for use in personal computers, workstations and servers. DIMMs (dual in-line memory modules) began to replace SIMMs (single in-line memory modules) as the predominant type of memory module as Intel P5-based Pentium processors began to gain market share.

[2]See Wikipedia

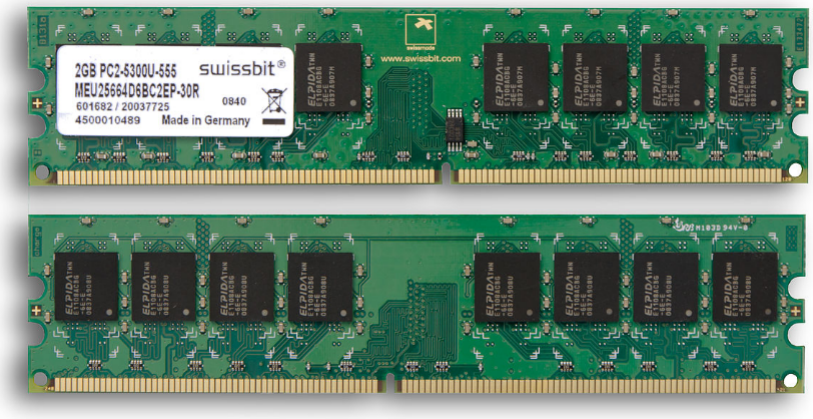


Figure 2: Dual Inline Memory Module

When directly accessing physical memory, the process needs to know *exactly* which memory addresses do and do not belong to it. The process can *see* all physical memory addresses but *should not* access addresses outside of its allocated region. There is no protection in this model to prevent one process from accessing kernel memory or memory for other processes.

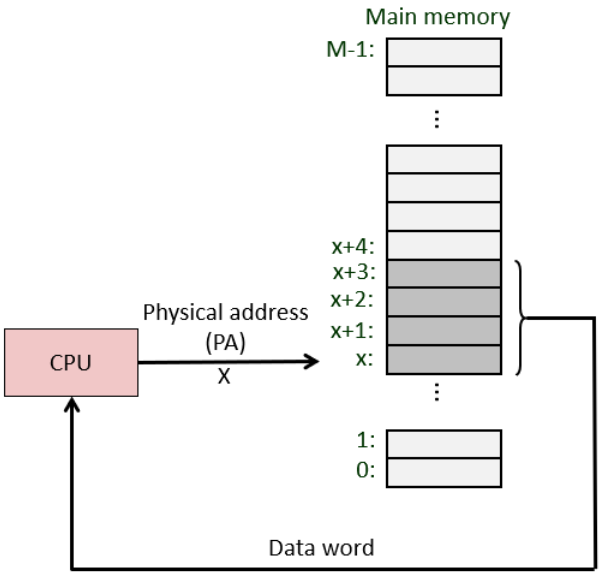


Figure 3: Using Physical Memory Directly

Allowing direct access to physical memory is risky. We will spend the next several lectures working to understand why this is so and ways in which the operating system can provide a process with a simplified view of memory that simplifies the programming model and removes the need for a process to be aware of other system users.

4.1 Issues

- Protection and Isolation. In this model, each process can access *all of physical memory* without restraint. This means that a programming error, or deliberate act, could result in the process accessing memory that is not part of its address space. No other process, or even the kernel, is safe from access, and possibly modification, by the process. At a minimum, users wishing to execute in this type of environment will want to know what other processes will be running at the same time. It is up to each individual user to safeguard their data as processes are not protected from each other.

- Relocation. If the process needs to know exactly which part of physical memory belongs to it, then that knowledge must be programmed into the executable. Moving the process to a new location, because of a different job mix or requirement changes, can be difficult. If direct memory addresses are used, as opposed to relative addresses[3], then moving the process to a new location is both cumbersome and time consuming.

- Process size. If the process grows, it is likely necessary to relocate it. If it shrinks, the unused space should be recovered for other uses. Both are difficult when processes directly access physical memory.

4.2 Impact on Programming Model

With physical memory, the process must be informed as to *where it lives*, that is, the address range(s), and each range must be contiguous! As you can imagine, this is complicated. The system must trust that the process will not access memory outside of its assigned region and the process must trust that no other process will access its memory region, *even if the processes have bugs!* Further, the number and size of processes that can run are wholly dependent on the amount of physical memory and the specific job mix. A changing job mix can be very difficult to accommodate.

Processes that grow and shrink dynamically are difficult to manage with this model. However, it works fine for a set of unchanging processes in a fully trusted environment.

4.3 Single or Multiple Address Ranges

When a process uses physical memory directly, it can be assigned a single address range or multiple address ranges. These physical address ranges are termed *segments*[4].

4.4 Strengths

The main benefit of using physical memory directly is speed. There is no latency as there is no translation when using physical memory directly.

4.5 Weaknesses

The principle problem with using physical memory directly is that in order for the process to be able to access any physical memory, *the process must be able to access all physical memory.* This

[3] See Wikipedia.

[4] segment: one of the parts into which something naturally separates or is divided; a division, portion, or section; 2) a portion of a program, often one that can be loaded and executed independently of other portions.

implies that any process can see all the code and data for all other processes in memory as well as the operating system. This means that processes cannot be isolated from each other. Consider trying to debug a program when you cannot be certain that the bug is from inside your code or the influence of another process (either their bug or deliberate action).

In other words, isolation is by agreement without any mechanism for enforcement. This is fine for a fully trusted environment (which are relatively rare) but not good for a general use computer system. It does not matter how well constructed your program is, unanticipated interactions with other processes can still cause problems, some of which can be nearly impossible to reproduce!

4.6 Additional Information

1. Before Memory Was Virtual, Peter J. Denning, George Mason University, 11/1/96.
2. What Every Programmer Should Know About Memory, Ulrich Drepper, Red Hat, Inc., November 21, 2007.

5 Address Space Management

The simplest way to manage memory is to let the processes manage it themselves. Tell each process what region of physical memory that it is allowed to access and then trust that the process will not access any memory outside of the agreed upon bounds.

5.1 Early Systems

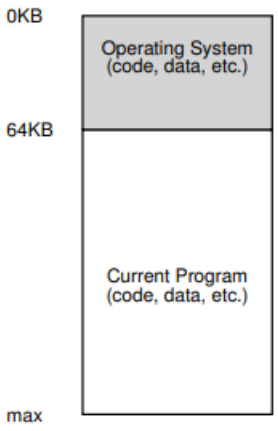


Figure 4: The Early Days

Yawn! That's it. The process and operating system share the entire physical memory. The smaller that the OS is (read: the less it does) then the larger the process can be. The process and operating system are not isolated in this model.

5.2 Multiprogramming

We will start our investigation of multiprogramming by looking at a simple representation of multiple processes in memory. Processes will occupy a single, fixed-sized segment. All segments

will be the same size, and the amount of physical memory will be an integer multiple of the process segment size.

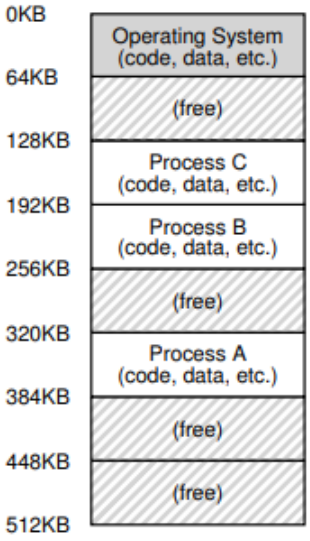


Figure 5: Simple Multiprogramming

In this model, each process is the *exact same size*.

- Benefits. If the size of physical memory is a multiple of the process size (common), then there is no external fragmentation. When one process ends, any new process will fix into that space.
- Issues. The size of *each process* must be the worst case; that is, each process gets the maximum amount of memory. There are usually very few truly large processes but many small ones (10% or less of the size of the worst case). This means that most of the space within almost all processes is wasted. This unused space is locked up in the memory region and cannot be used by any other process. Further, since most processes are interactive, we need as many processes in memory as possible in order to increase the chance of there being a process in the `RUNNABLE` state. *Keeping the CPU busy is very important and inefficient use of memory hampers that effort.*

5.3 Isolation Principle

Isolation is a key principle in building reliable systems. If two entities are properly isolated from each other, then one can fail without affecting the other. Operating systems strive to isolate processes from each other and in this way prevent one from harming any other, either by accident or by design. By using memory isolation, the OS further ensures that running programs cannot affect the operation of the underlying OS. Some modern OS's take isolation even further, by walling off some parts of the OS from itself. Such microkernels and exo-kernels may provide greater reliability than typical monolithic kernel designs[5].

[5]These are beyond the scope of this class.

5.4 Address Space Virtualization

5.4.1 Benefits

Benefit	Description
Improved Memory Management	Each process gets the same uniform linear address space.
Relocation	Since the process does not *see* where its code and data are located in physical memory, relocation is much easier. The virtual-to-physical translation process will *know* where to find the different parts of the process (see MMU). Note that while the process *sees* a contiguous address space, the underlying assignment of physical memory is usually fragmented.
Process Size	The process is now limited by the size of the virtual address space, not the available physical memory. Each process has the same maximum size.
Efficiency	Use physical memory as a cache for parts of a virtual address space, swap space for the rest. By keeping less of a process in memory, we are able to keep more total processes in memory. This increases the chance of the scheduler finding a process in the runnable (ready) state.
Isolation	One process can't interfere with the memory of another. User program cannot access privileged kernel information and code. With VM, the process cannot access any address that is not part of its address space. Typically, for an architecture that is k-bits, the process address range will be from 0 to $2^k - 1$. This means that *all possible* (virtual) addresses are within the process space. Any one process cannot access a different process or kernel space unless specific steps are taken by both participants to share a memory region.

Table 2: Benefits of Address Space Virtualization

5.4.2 Virtualization Goals

Goal	Description
Transparency	The OS should implement virtual memory in a way that is invisible to the running program. Thus, the program should not be aware of the fact that memory is virtualized; rather, the program behaves as if it has its own private physical memory. Behind the scenes, the OS (and hardware) does all the work to multiplex memory among many different jobs, and hence implements the illusion.

Goal	Description
Efficiency	The OS should strive to make the virtualization as efficient as possible, both in terms of time (not making programs run much more slowly) and space (not using too much memory for structures needed to support virtualization). In implementing time-efficient virtualization, the OS will have to rely on hardware support, including hardware features such as the Memory Management Unit (MMU).
Protection	The OS should make sure to protect processes from one another as well as the OS itself from processes. When one process performs a load, a store, or an instruction fetch, it should not be able to access or affect in any way the memory contents of any other process or the OS itself (anything outside its address space). Protection thus enables us to deliver the property of isolation among processes; each process should be running in its own isolated cocoon, safe from the ravages of other faulty or even malicious processes.

Table 3: Virtualization Goals

The improved memory management uses a fixed model for all processes. Each process can now use any possible virtual address and still not access the address space of a different process. Within the address space, the process only needs to be concerned with accessing *allocated* (e.g., valid) addresses as well as avoiding the protected memory region allocated to the operating system.

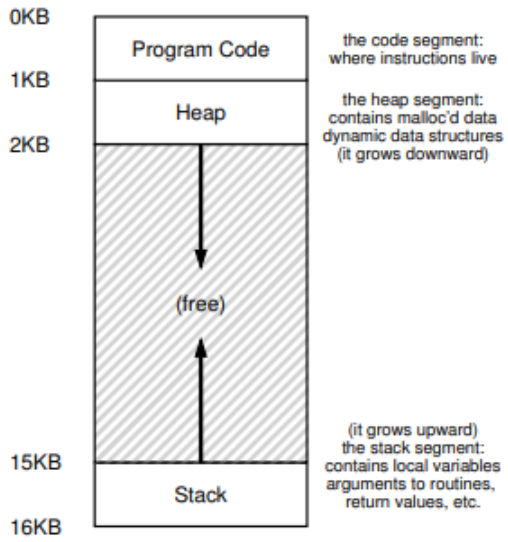


Figure 6: Process View of Memory

With virtual memory, the process is assigned a contiguous region that it cannot escape. This region is then mapped to physical memory based on OS policy. The main techniques are paging and segmentation.

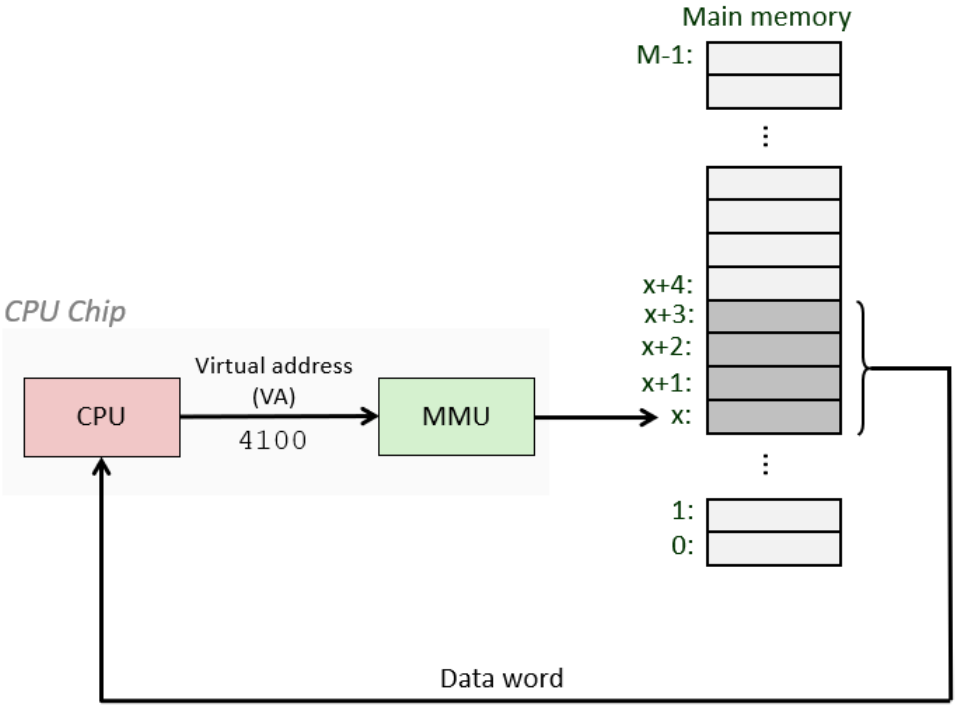


Figure 7: Using Virtual Memory

5.4.3 Impact on Programming Model

The programming model is now greatly simplified. The process only sees its own address space and can use as little or as much of it as required. Knowledge of memory use by other processes, and even the kernel, is hidden.

The process sees a *flat* or *linear* address space and does not need to be concerned with how physical memory and disk space is organized to support this view. A linear memory model refers to a memory addressing paradigm in which *memory appears to the process as a single contiguous address space.*

> Remember that physical memory is hierarchical. There is the register file, various levels and types of caches, main memory, disks, and perhaps network resources, all as part of the memory hierarchy. However, the programming model hides all this complexity. This frees the programmer so that they only have to go beyond the linear memory model when necessary, usually for performance.

Memory management and address translation can still be implemented on top of a flat memory model in order to facilitate operating systems functionality, resource protection, multitasking, or to increase the memory capacity beyond the limits imposed by the processor's physical address space, but the key feature of a flat memory model is that the entire memory space is linear, sequential and contiguous from address zero to `MaxBytes - 1`[6].

[6]See Wikipedia.

6 Virtual Address Translation

Now that we have virtual memory and virtual memory addresses, how do we translate this into the necessary physical memory addresses? Can the physical addresses change over time, without the virtual address changing, and, if so, how do we manage this?

Since addresses need translation frequently[7], specialized hardware called the Memory Management Unit MMU (See §6.1) exists to speed up the translation. Depending on the virtual memory model being used, the MMU can be simple or quite complex.

With address translation, the hardware transforms each memory access (e.g., an instruction fetch, load, or store), changing the virtual address provided by the instruction to a physical address where the desired information is actually located[8]. Thus, on each and every memory reference, an address translation is performed by the hardware to redirect application memory references to their actual locations in physical memory.

Hardware alone cannot manage all aspects of virtual memory. The operating system needs to be involved to ensure that the translation is correct in all cases. This type of *memory management* keeps track of which parts of physical memory are free, or unused, and how a currently executing process address space maps to physical memory.

A key feature of virtual memory is that the process believes it has exclusive access to all system memory. The operating system supports this illusion by managing the *shared* physical memory in a way that enables this view of memory. Through virtualization, the OS, with help from hardware, turns the ugly machine reality into a useful, powerful, and easy to use abstraction.

6.1 The Memory Management Unit

A memory management unit (MMU)[9] is a computer hardware device that processes all memory references, performing the translation of virtual memory addresses to physical addresses. An MMU effectively performs virtual memory management, handling at the same time memory protection, cache control, and bus arbitration.

As a program runs, the memory address that it uses to reference its data is a virtual (logical) address. The real-time translation to the physical address is performed in hardware by the CPU's Memory Management Unit (MMU). The MMU has two special registers that are accessed by the CPU's control unit. Datum to be sent to main memory or retrieved from memory is stored in the Memory Data Register (MDR). The desired logical memory address is stored in the Memory Address Register (MAR). The address translation is also called address binding and uses a memory map that is programmed by the operating system.

A key job of the operating system is to load the appropriate data into the MMU when a processes is made active on a processor and to respond to the occasional page faults by loading the needed memory and updating the memory map. This data is stored in the process context.

[7] At least one translation per instruction.
[8] Eventually, this location must be an address in main memory.
[9] See Wikipedia.

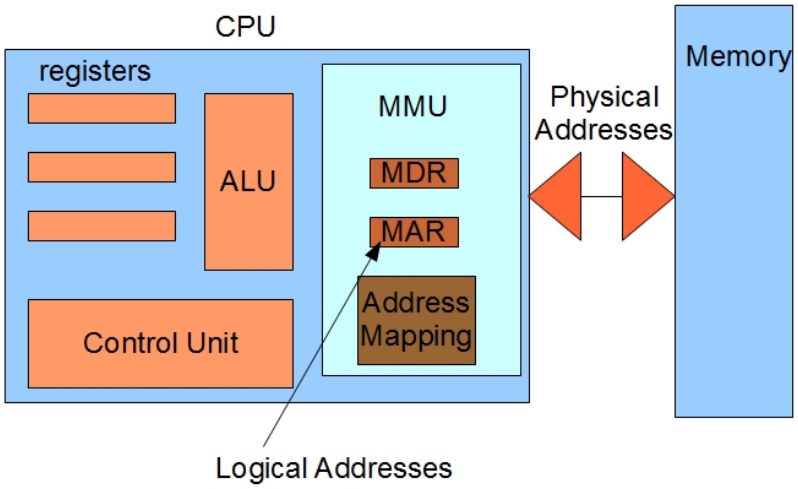


Figure 8: MMU Overview

Inside the MMU is a table, of some sort, that aids translation. It may be implemented in a variety of ways, including a CAM, linked list, array, hash table, etc.

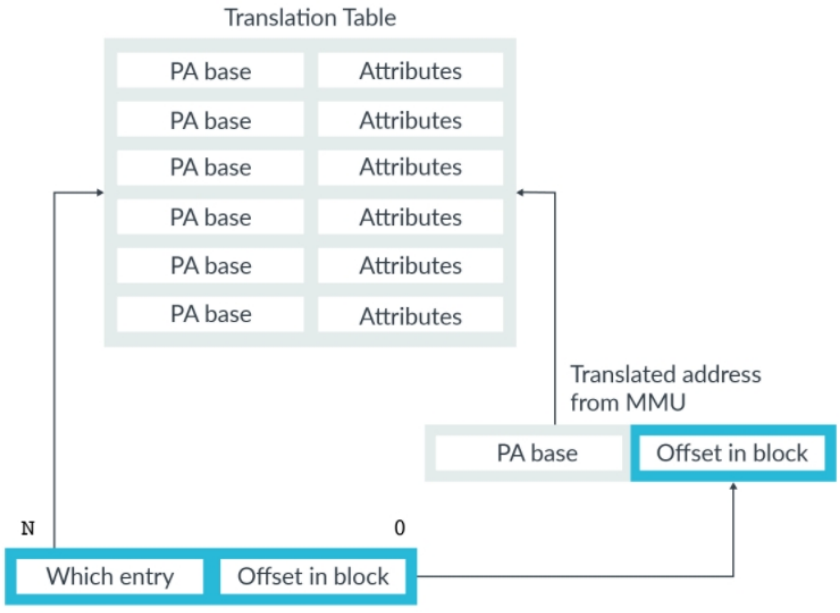


Figure 9: Generic Address Translation

When we get to paged virtual memory, we will see that the MMU will make use of a specialized memory, the translation lookaside buffer, as a cache to speed address translation.

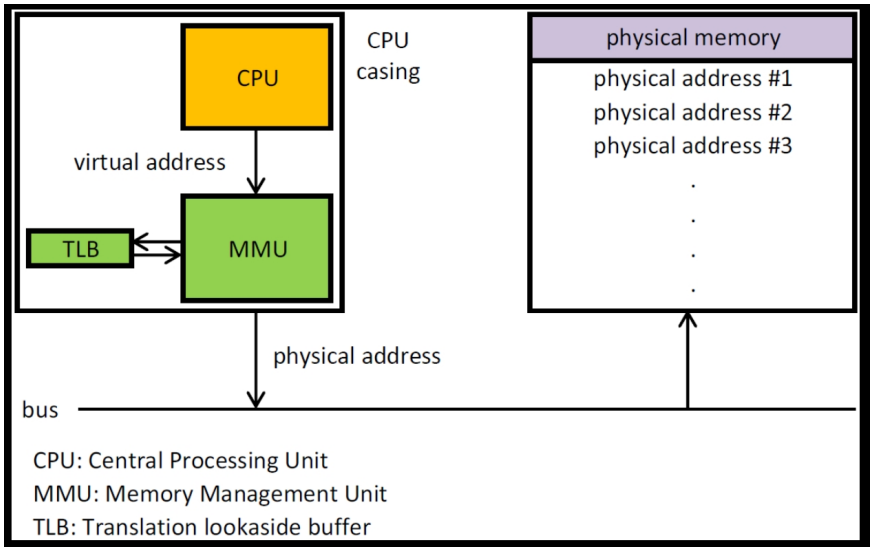


Figure 10: MMU – paging

6.1.1 Intel x86 MMU

The x86 architecture has evolved over a very long time while maintaining full software compatibility, even for OS code. Thus, the MMU is extremely complex, with many different possible operating modes. The IA-32 MMU is described here.

The CPU primarily divides memory into 4 KB pages. Segment registers, fundamental to the older 8088 and 80286 MMU designs, are not used in modern OSes, with one major exception: access to thread-specific data for applications or CPU-specific data for OS kernels, which is done with explicit use of the FS and GS segment registers. All memory access involves a segment register, chosen according to the code being executed. The segment register acts as an index into a table, which provides an offset to be added to the virtual address. Except when using FS or GS, the OS ensures that the offset will be zero.

After the offset is added, the address is masked to be no larger than 32 bits. The result may be looked up via a tree-structured page table, with the bits of the address being split as follows: 10 bits for the branch of the tree, 10 bits for the leaves of the branch, and the 12 lowest bits being directly copied to the result. Some operating systems, such as OpenBSD with its W^X feature, and Linux with the Exec Shield or PaX patches, may also limit the length of the code segment, as specified by the CS register, to disallow execution of code in modifiable regions of the address space.

Minor revisions of the MMU introduced with the Pentium have allowed very large 4 MB pages by skipping the bottom level of the tree (this leaves 10 bits for indexing the first level of page hierarchy with the remaining $10 + 12$ bits being directly copied to the result). Minor revisions of the MMU introduced with the Pentium Pro introduced the physical address extension (PAE) feature, enabling 36-bit physical addresses with $2 + 9 + 9$ bits for three-level page tables and 12 lowest bits being directly copied to the result. Large pages (2 MB) are also available by skipping the bottom level of the tree (resulting in $2 + 9$ bits for two-level table hierarchy and the remaining 9+12 lowest bits copied directly). In addition, the page attribute table allowed specification of cacheability by looking up a few high bits in a small on-CPU table.

No-execute support was originally only provided on a per-segment basis, making it very awk-

ward to use. More recent x86 chips provide a per-page no-execute bit in the PAE mode. The W^X, Exec Shield, and PaX mechanisms described above emulate per-page non-execute support on machines x86 processors lacking the NX bit by setting the length of the code segment, with a performance loss and a reduction in the available address space.

6.1.2 The ARM Architecture

The ARM architecture focuses on paging. There is support for segmentation.

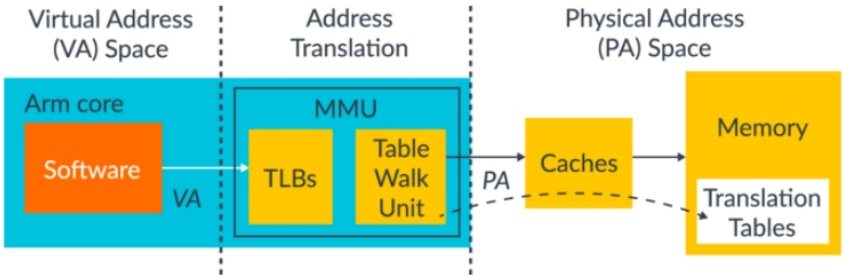


Figure 11: ARM Address Translation

6.2 Dynamic Relocation

With a single register in the MMU, our simple process model now has *dynamic relocation*[10]. This *base register* contains the address in physical memory where our process is located. The virtual addresses within the process are now *relative* to this base – "relative addressing". Thus, address translation is simply the equation:

$$PA = base + VA$$

where PA stands for "physical address" and VA stands for "virtual address". The base register now becomes part of the process context.

[10]Dynamic relocation is where data currently stored at one location in the computer memory is relocated to another part of the memory transparently to the process accessing the data by use of "base plus offset" addressing.

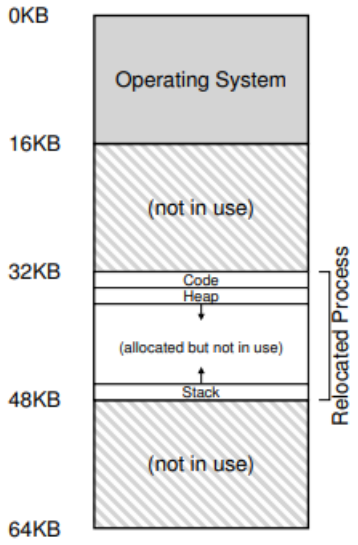


Figure 12: Physical Memory with a Single Relocated Process

6.3 Hardware Support

In order to allow variable sized process spaces, and thus make more efficient use of physical memory, each process will now have a pair of registers in our MMU called the *base register* and the *bounds register*, also called the *limit register* depending on how it is used. This new bounds register will help the MMU to determine if the virtual address is outside of the process's "memory chunk", called the process *segment*. Variable sized memory regions, called *segments*, help with the problem of internal fragmentation but make the problem of external fragmentation very problematic.

This changes the formula for address translation. Two forms are commonly used.

```
PA = base + VA
if NOT (PA <= bounds_register) then
segmentation_violation

// or

if NOT (VA <= bounds_register) then
segmentation_violation
PA = base + VA
```

Note that there is no need to check against the base register. The virtual address cannot be negative. In fact, *all memory addresses are unsigned.*

6.4 Relocation Redux

Now that we have variable sized segments, relocation has become complicated.

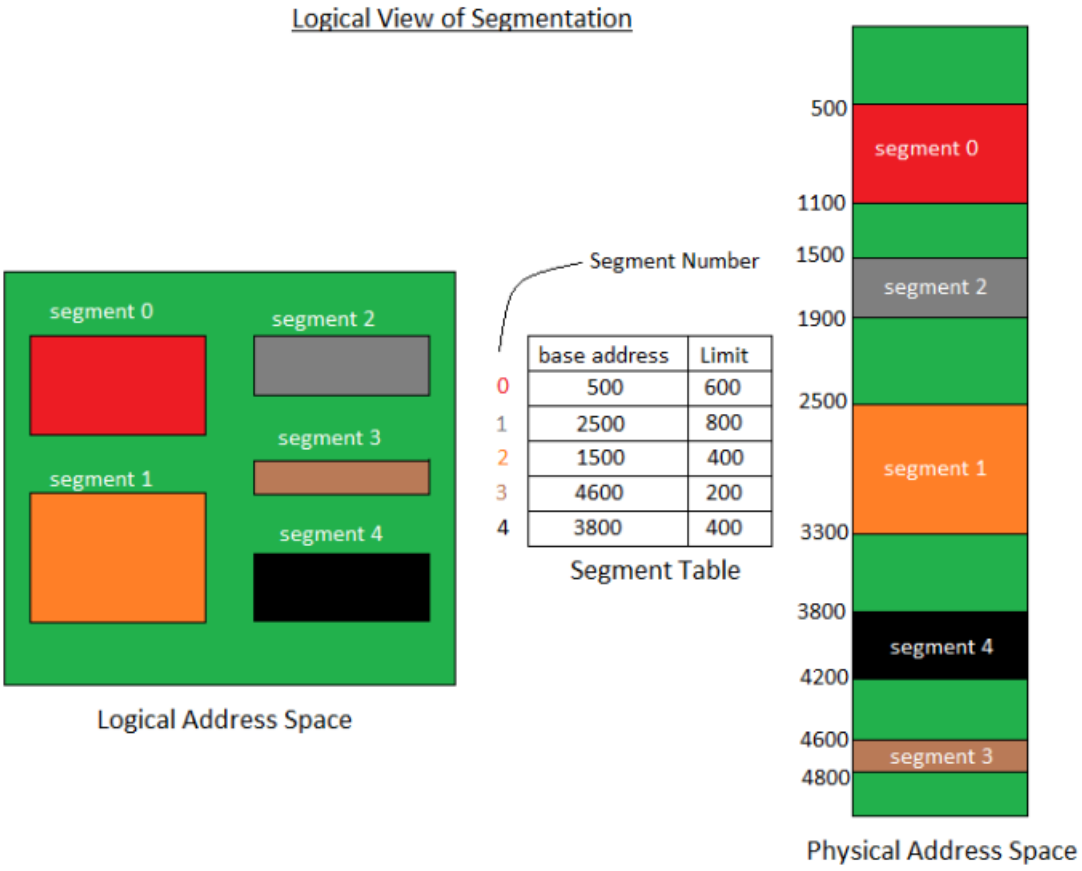


Figure 13: Multiple Variable-Sized Segments

6.4.1 Finding the Right Hole

Unused memory segments ("holes") are now variable sized and we are unlikely to find one that is exactly the size that we need. How do we choose which hole is right for our process? Any strategy will invariably search a data structure containing information about all the holes. We know that this approach will be $\mathcal{O}(n)$ where n is the number of holes, which is variable.

6.4.2 Memory Compaction

What if there are many small, but unsuitable, holes? It would be great to move the all together into one large hole. This is the purpose of memory compaction. However, while the compaction of the in-use segments is occurring, we usually can't let any process run, so the system comes to a standstill until the copying is complete. If run to completion, the end result will be all in-use segments at one end of memory and one big hole occupying the rest of memory.

6.5 Operating System Issues

We now have several issues:

1. Less internal fragmentation at the cost of introducing external fragmentation
2. A more complex process context
3. A more complex (read: slower) address translation process
4. More OS state for tracking holes, in addition to in-use segments.

There is still more we can do about fragmentation and also about the holes. We will drop the idea of using a single memory chunk for each process and explore breaking the process into parts. There are several advantages to this strategy and, surprisingly, one of them helps quite a bit with internal fragmentation.

At this point, we have the concept of *virtual memory*. The CPU emits virtual memory addresses instead of physical memory addresses. The MMU translates between virtual and physical addresses. Don't forget, we still require that the entire process fit into physical memory.

7 Memory API

7.1 The Stack

In running a C program, there are two types of memory that are allocated. The first is called stack memory, and allocations and deallocations of it are managed implicitly by the compiler for you, the programmer; for this reason it is sometimes called *automatic memory*.

Declaring memory on the stack in C is easy. For example, let's say you need some space in a function `func()` for an integer, called x. To declare such a piece of memory, you just do something like this:

```
void func() {
    int x; // creates room for one 32-bit integer
           // on the stack and uses ‘‘x’’ as the reference
    .
    .
    .
}
```

The compiler does the rest, making sure to make space on the stack when you call into `func()`. When you return from the function, the system deallocates the memory for you when the stack frame is deleted. Thus, if you want some information to live beyond the call invocation, you had better not leave that information in the local stack frame.

7.2 The Heap

It is the need for long-lived memory that gets us to the second type of memory, called heap memory, where all allocations and deallocations are explicitly handled by you, the programmer. A heavy responsibility, no doubt! And certainly the cause of many bugs. But if you are careful and pay attention, you will use such interfaces correctly and without too much trouble. Here is an example of how one might allocate an integer on the heap:

```
void func() {
    int *x = (int *) malloc(sizeof(int));
    .
    .
    .
}
```

A couple of notes about this small code snippet. First, you might notice that both stack and heap allocation occur on this line: first the compiler knows to make room for a pointer to an integer when it sees your declaration of said pointer (`int *x`); subsequently, when the program calls `malloc()`, it requests space for an integer on the heap; the routine returns the address of such an integer (upon success, or NULL on failure), which is then stored on the stack for use by the program. Because of its explicit nature, and because of its more varied usage, heap memory presents more challenges to both users and systems.

7.3 malloc()

The single parameter `malloc()` takes is of type `size_t` which simply describes how many bytes you need. However, most programmers do not type in a number here directly (such as 10); indeed, it would be considered poor form to do so. Instead, various routines and macros are utilized. For example, to allocate space for a double-precision floating point value, you simply do this:

```
double *d = (double *) malloc(sizeof(double));
// or
double *d = (malloc(sizeof(double));
```

When declaring space for a string, use the following idiom: `malloc(strlen(s) + 1)`, which gets the length of the string using the function `strlen()`, and adds 1 to it in order to make room for the end-of-string character (NULL).

Casting doesn't really accomplish anything, other than tell the compiler and other programmers who might be reading your code: "yeah, I know what I'm doing." By casting the result of `malloc()`, the programmer is just giving some reassurance; the cast is not needed for the correctness.

7.4 free()

To free heap memory that is no longer in use, programmers simply call `free()`:

```
int *x = malloc(10 * sizeof(int));
    .
    .
    .
free(x);
```

The routine takes one argument, a pointer returned by `malloc()`. Thus, you might notice, the size of the allocated region is not passed in by the user and must be tracked by the memory-allocation library itself.

7.5 Common Errors

7.5.1 Forgetting To Allocate Memory

Many routines expect memory to be allocated before you call them. For example, the routine `strcpy(dst, src)` copies a string from a source pointer to a destination pointer. However, if you are not careful, you might do this:

```
char *src = "hello";
char *dst; // oops! unallocated
strcpy(dst, src); // segfault and die
```

When you run this code, it will likely lead to a segmentation fault, which will cause your process to terminate.

In this case, the proper code might instead look like this:

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src) + 1);  // account for NULL terminator
strcpy(dst, src); // works properly
```

7.5.2 Not Allocating Enough Memory

A related error is not allocating enough memory, sometimes called a buffer overflow. In the example above, a common error is to make almost enough room for the destination buffer.

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src)); // too small!
strcpy(dst, src); // not what you expected
```

Oddly enough, depending on how `malloc()` is implemented and many other details, this program will often run seemingly correctly. In some cases, when the string copy executes, it writes one byte past the end of the allocated space, but in some cases this is harmless, perhaps overwriting a variable that isn't used anymore. These overflows, however, can be incredibly harmful and in fact are the source of many security vulnerabilities in systems.

For the above code snippet, try to predict what will happen when the string `dst` is printed.

7.5.3 Forgetting to Initialize Allocated Memory

With this error, you call `malloc()` properly, but forget to fill in some values into your newly-allocated data type. Don't do this! If you do forget, your program will eventually encounter an uninitialized read, where it reads from the heap some data of unknown value. Who knows what might be in there? If you're lucky, some value such that the program still works (e.g., zero). If you're not lucky, something random and harmful. Many systems zero memory on allocation, but not all. **Programmer beware!**

7.5.4 Forgetting To Free Memory

Another common error is known as a memory leak, and it occurs when you forget to free memory. In long-running applications or systems (such as the OS itself), this is a huge problem, as slowly leaking memory eventually leads one to run out of memory, at which point a restart is required. Thus, in general, when you are done with a chunk of memory, you should make sure to free it.

Note that using a garbage-collected language doesn't help here: if you still have a reference to some chunk of memory, no garbage collector will ever free it, and thus memory leaks remain a problem even in more modern languages.

In some cases, it may seem like not calling `free()` is reasonable. For example, your program is short-lived, and will soon exit; in this case, when the process dies, the OS will clean up all of its allocated pages and thus no memory leak will take place per se. While this certainly "works", it is probably a bad habit to develop, so be wary of choosing such a strategy. Always free memory that you have allocated and then it is less likely you will have a memory leak.

7.5.5 Freeing Memory Before You Are Done With It

Sometimes a program will free memory before it is finished using it; such a mistake is called a dangling pointer, and it, as you can guess, is also a bad thing. The subsequent use can crash the program, or overwrite valid memory (e.g., you called `free()`, but then called `malloc()` again to allocate something else, which then recycles the errantly-freed memory).

7.5.6 Freeing Memory Repeatedly

Programs also sometimes free memory more than once; this is known as the double free. The result of doing so is undefined. As you can imagine, the memory–allocation library might get confused and do all sorts of weird things; crashes are a common outcome.

7.5.7 Calling free() Incorrectly

One last problem we discuss is the call of `free()` incorrectly. After all, free() expects you only to pass to it one of the pointers you received from `malloc()` earlier. When you pass in some other value, bad things can (and do) happen. Thus, such invalid frees are dangerous and of course should also be avoided.

7.5.8 Memory Leak Detection

As you can see, there are lots of ways to abuse memory. Because of frequent errors with memory, a whole ecosphere of tools have developed to help find such problems in your code. Check out both purify and valgrind; both are excellent at helping you locate the source of your memory-related problems. Once you become accustomed to using these powerful tools, you will wonder how you survived without them.

7.6 How Does `malloc()` Actually Work?

The system call `malloc()` allocates space from the heap (`sbrk() makes the heap bigger as necessary`). Malloc needs to track the size of the space so that when `free()` is called that this space can be freed in its entirety. The approach is to trade (*overhead*) for convenience when allocating a block of memory. The `malloc()` library prepends a header to the memory block that contains all the information necessary to manage and free this block. The *header* is fixed-size so the start of the *actual block* of memory can be calculated from the address that `malloc()` returned to the calling program.

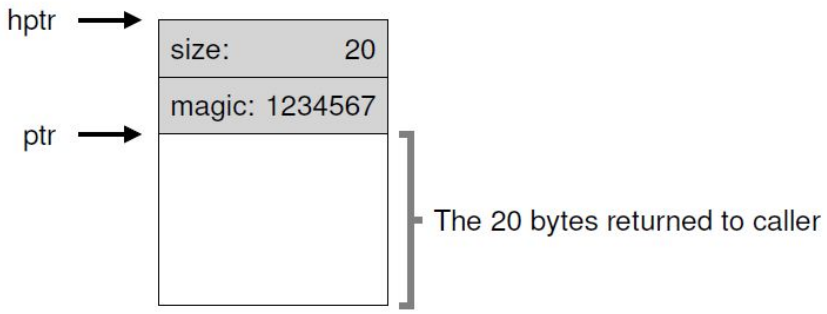


Figure 17.2: **Specific Contents Of The Header**

How does `malloc()` track the free space inside an already allocated region (e.g., the heap)? The technique used is to embed the list within the region itself.

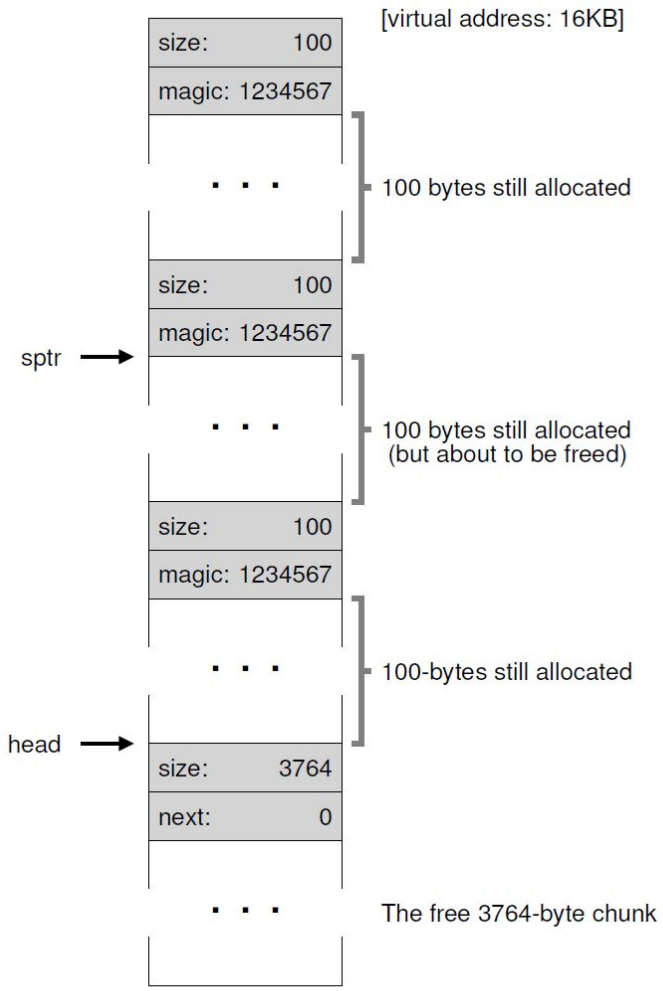


Figure 17.5: **Free Space With Three Chunks Allocated**

```
typedef __node_t {
    int size;
    struct __node_t next;
} node_t;
```

7.7 Other Memory Allocators

- `calloc()`. Allocates memory for the stated number of elements each of the requested size in bytes (e.g., number_of_elements * size_of_each_element). The memory is set to zero.
- `realloc()`. Changes the size of the memory block pointed to by the supplied pointer. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory *will not* be initialized.
- `alloca()`. Allocates memory in the local stack frame. This has the side effect that the memory is automatically freed on return from the function. No call to `free()` is required.

7.8 Additional Reading

1. *A Malloc Tutorial*, Marwan Burelle, Laboratoire Systeme et Securite de l'EPITA (LSE), February 16, 2009.

8 Segmented Virtual Memory

Segmentation is a memory management technique where process memory is divided into sections called *segments*. In a computer system using segmentation, a reference to a memory location includes a value that identifies a segment and an offset (memory location) within that segment[11].

Thus far, each process has one segment (think *block*) that can be variable-size. Initially, each segment was the same size, so *external fragmentation* was minimized. But since the segment size had to be some *worst-case*, we had a large problem with *internal fragmentation*. The solution was to make each segment of variable size.

This is accomplished through the use of *base and bounds* or *base and limit* registers that are now private to each process space and used in the MMU. This helped some with internal fragmentation but made external fragmentation problematic. When one process exits, there is no guarantee that a new process will have a segment that fits into the space freed. Oh yeah, how do we allocate these "holes" – first-fit, best-fit, worst-fit?

One solution to the external fragmentation problem is to periodically *compact* memory. That is, move all the used space to one end and all the unused space to the other end. Lots and lots of copy operations (move from memory into CPU then move from CPU to new memory location). This means lots and lots of time with the CPU not working on user processes. How often do you compress? Do all processes have to be modified to compress memory? Is there a selection strategy (e.g., more overhead)? Compress all or "just enough for now"?

So now we have some hardware help in terms of an MMU which contains a single base-bounds register pair. This pair must be saved/restored on a context switch. Let's consider the amount of space between the stack and heap inside the process (internal fragmentation).

We can do better.

[11] See Wikipedia.

8.1 Intel Segment Registers

- CS, code segment. Machine instructions exist at some offset into a code segment. The segment address of the code segment of the currently executing instruction is contained in CS.
- DS, data segment. Variables and other data exist at some offset into a data segment. There may be many data segments, but the CPU may only use one at a time, by placing the segment address of that segment in register DS.
- SS, stack segment. The stack is a very important component of the CPU used for temporary storage of data and addresses. Therefore, the stack has a segment address, which is contained in register SS.
- ES, extra segment. The extra segment is exactly that: a spare segment that may be used for specifying a location in memory. There are two additional extra segments, called FS and GS. Names FS and GS come from the fact that they were created after ES: E, F, G. They exist only in the 386 and later x86 CPUs.
- Extra segments ES, FS, and GS can be used for both data or code.

The IA32 and x86-64 architectures allow up to six active segments. You can have more than six segments (all segment descriptors are in the descriptor table), but the programmer will have to manage segment loads and unloads manually. A segment becomes active by loading its descriptor into one of the segment registers. In theory, a program can have millions of segments. We will revisit this idea under paging.

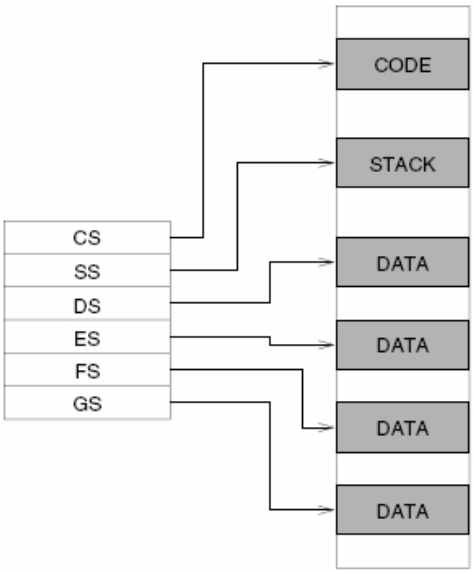


Figure 14: Intel's 6 Segment Registers

8.2 Fixed-sized Segments

With fixed-sized segments, memory can be partitioned into N pieces, each of which is the same size. If we limit physical memory to being an integer multiple of the segment size, then the number of segments that fit into memory is easily calculable as is the location of each segment. Each segment can then be marked as *free* or *in use* and this is the only free space management that is required. However, as we have seen, fixed-sized segments are far from optimal.

8.3 Variable-sized Segments

With variable-sized segments, memory is not so easily partitioned and thus a more complex method for tracking and allocating memory to segments is required. This method is required whether or not a process is comprised of as little as one or even many variable-sized partitions. Since the standard term for a "partition" is *segment*, we will adopt that term from here on out.

8.4 Minimizing Fragmentation

For now, we will assume that variable-sized segmentation is sufficient for addressing *internal fragmentation* (hint: it's not) and deal with *external fragmentation*.

Part of the problem with variable-sized segments is that when they are freed (deallocated), they leave behind variable-sized *holes*. As segments are allocated and freed, our free memory becomes more and more fragmented, finding a suitable hole for allocation to a segment becomes a much harder task. At some point, there may be enough memory to satisfy an allocation request but the request cannot be satisfied because it isn't available in a large enough *contiguous* block. This leads to the very expensive *compaction* problem. If a system doesn't have a lot of physical memory and isn't very busy, this may not be burdensome. However, for most systems, compaction represents a major problem and one for which it is worth developing strategies to address.

8.5 Multiple Segments

If we let each *part* of our process occupy a segment by itself and thus allow the entire process to be composed of many segments, we can achieve better memory utilization. So now, each process has *many base/bounds register pairs* – more hardware support. Let's put it into the MMU. In the Intel X86 architecture, we have segment registers cs, ss, ds, es, fs, and gs. For variable sized segments, we will have a bounds, or limit, register in addition to the base register for each active segment.

The MMU now supports a process containing many, variable-length segments. More hardware support but more work on a context switch. Did it buy us anything?

For large address spaces, only a fraction is used. Our new model helps greatly with internal fragmentation.

1. The code segment size is known at compile time.
2. The heap and stack segments can be given "reasonable" default sizes that suits *most* processes. May have compiler support.
3. If a stack or heap needs to grow:
   (a) Is the space "next" to the top of the heap or bottom of the stack free? Change the bounds.

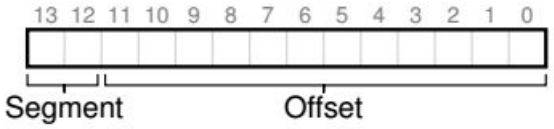
(b) Worst case is that the segment is copied to a larger segment and the old segment freed. Ugh.

8.6 Segmented Address Translation Example

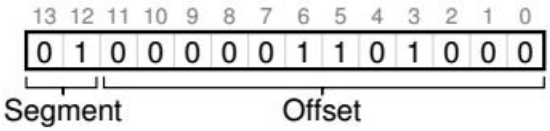
This example uses a 14-bit address space. The example is taken from the OSTEP text.

8.6.1 Segment Identifier and Offset

How do we know which segment? Steal bits from the virtual address. Why the high order bits?



(a) 2-bits for Segment Selector



(b) VA 4200

Figure 15: Address Layout for Example

Segment	Selector	Base	Size	GP?	GN?
code	00	32KB	2K	1	0
data	01	34KB	2K	1	0
stack	11	28KB	2K	0	1

Table 4: Segment Information for Example

Virtual	Physical
100_{10}	$??_{10}$
4200_{10}	$??_{10}$

Table 5: Translation Practice

$$\text{PA} = (-1) * \text{GN} * \text{ MSS} + \text{Base} + \text{Offset}$$

Figure 16: Segment Address Translation Formula

The formula of Figure 16 works correctly regardless of the direction that the segment grows. This seems much simpler than the two separate techniques of the OSTEP text.

8.6.2 The Stack

The stack grows downward, which slightly complicates our address math. The stack is allocated at physical address 28KB which is the *bottom* of the stack. The stack grows *downward* towards 26KB.

Let's access an address in our stack: VA 15KB which maps to PA 27KB

```
15KB: 11 1100 0000 0000  (0x3C00)
Segment Selector: 11
Offset: 1100 0000 0000  (0xC00, $3072_{10}$)

Max Segment Size (MSS): 1111 1111 1111 (4KB, $4096_{10}$)
```

Recall the formula from Figure 16:

$$\text{PA} = (-1) * \text{GN} * \text{MSS} + 28\text{KB} + \text{Offset}$$

Then for this example,

MSS is 4KB
Offset is 3KB (3072) which half the way into the current stack?
$\text{PA} = (-1) * 1 * 4096 + 28672 + 3072$
PA = 27648 or 27KB

We now see how to find the correct address in the stack segment. For our example, virtual address 15KB is located at physical address 27KB.

8.7 More Stuff For the MMU

We now have the MMU doing a lot of work for address translation. It turns out that we would also like to know what we can do to a memory location. For example, in most programming languages, we do not want the code segment to be modified during execution. Interestingly, under our segmentation model the type of protection we want can be specified at the *segment level* and so only adds a few bits of overhead for this very valuable capability.

8.8 One Little Heap Problem

Remember how we can allocate memory in the heap and then also deallocate it? It is possible for our heap to become quite *sparse*. With our segment model, we can't deal with this. Do we really want to periodically compact a heap and update all those memory addresses in our program? This is pretty ugly.

8.9 Compaction

We still haven't managed to solve the compaction problem. It is a very serious problem that can greatly affect performance. Since the OS can't know when it will need to compact, we must assume it will be at the most inconvenient time. Next, we will look at a mechanism that all but does away with both internal and external fragmentation, but has problems of its own.

The following example is based on drawings from lwn.net.

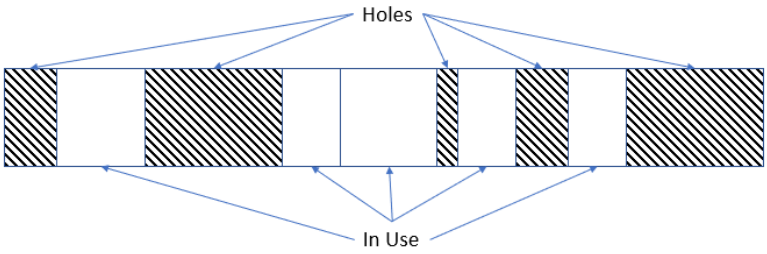


Figure 17: In Need of Compaction

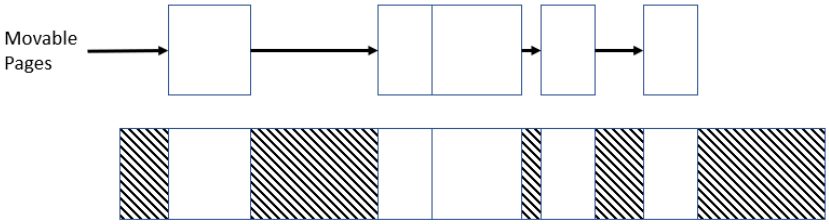


Figure 18: Identify Regions for Relocation

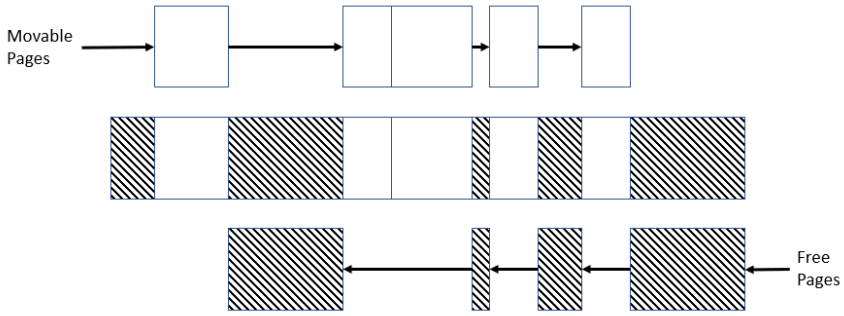


Figure 19: Identify Holes to Move

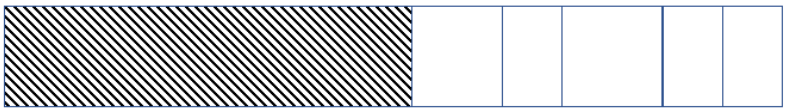


Figure 20: Compaction Complete

9 Free Space

Memory that is not *allocated* is said to be *free*. The management of *free space* is critical to overall performance of the segmented memory model.

9.1 Free List Management

Physical memory can be viewed as a byte-array that is initially contiguous and free. Any allocation of this memory results in the splitting of memory. Since the lifetime of some sections of allocated memory will be longer than others, it is reasonable to assume that holes will appear over time. Nothing can really be done about these holes except track them and possibly combine adjacent ones.

One way to track free space is to use a simple linked list. Each node contains the start address and length of the *hole*.

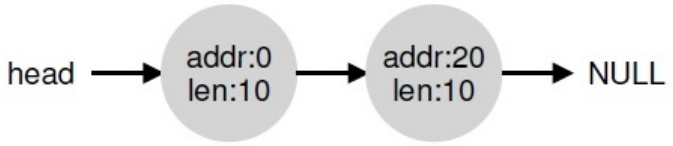


Figure 21: Using A Linked List for Tracking Free Space

Not bad. However, there are some problems, including how to traverse the list and what to do if there are two holes that are adjacent. If we use a simple linked list, then we will always traverse from the head of the list. Over time, it will take longer and longer to find a suitable hole. When two holes are adjacent, we can choose to combine them in a technique called *coalescing*. The technique is fairly simple but requires a check on every deallocation. And finally, there is the problem of how much of a hole to allocate. Too many small holes and the overhead is quite large. Do we allocate entire holes or partial holes? If partial holes, do we allocate from the beginning or end of the hole? Does it matter? Does it impact coalescing?

9.2 Allocation Strategies

- Best fit $\mathcal{O}(n)$
- First fit $\mathcal{O}(n)$
- Buddy Allocator $\mathcal{O}(1)$
- Worst fit $\mathcal{O}(n)$
- Next fit $\mathcal{O}(n)$

Figure 22: The Buddy Allocator by Amie Roten

Fun Facts about Buddy Allocation

BASICS:

Buddy allocation is an allocation scheme often used for **heap** management and allocation (think 'malloc()')

Other allocation/memory management schemes we discussed this term included a linear linked list schema and a circular linked list schema. These data structures track the free space/'holes' in memory that can be allocated for new data.

For the linked list-style free lists, several different policies exist to select a 'chunk' of free space, including first fit, next fit, best fit, and worst fit. These approaches all intend to reduce the amount of **external fragmentation**, but give O(n) performance, since the whole list may need to be traversed to find the most appropriate free segment.

Buddy allocation, however, has O(1) performance, and once memory became cheap, a schema that prioritizes time-efficiency over space-efficiency became more desirable.

DATA STRUCTURE:

The buddy allocator consists of an **array of free lists**, each corresponding to a size (2^n bytes).

The heap can be generally thought of as a single block of the largest possible size that gets 'split' as smaller blocks are requested, although other algorithms exist to pre-populate common block sizes' lists.

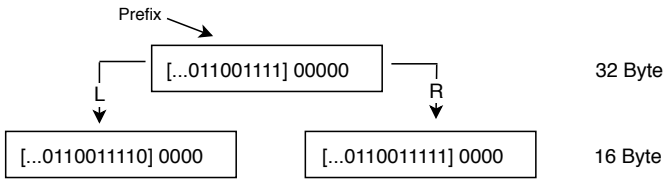
REQUEST PROCESS:

When a routine makes a call to malloc(), they provide the desired size needed to accommodate their data. If the size needed is **NOT** a power of 2 (not an exact size of a block), the request is rounded up to the next largest power of 2. (i.e., a request of 3.2kb would be rounded up to 4kb).

If there exists a free block in that list, malloc() will allocate the block and return a pointer to that free block to the calling routine. However, if there is not a free block of the correct size, one must be procured through one of the following procedures. *note: this is for the BINARY BUDDY ALLOCATOR. Others do exist, but not discussed in class.

SPLITTING:

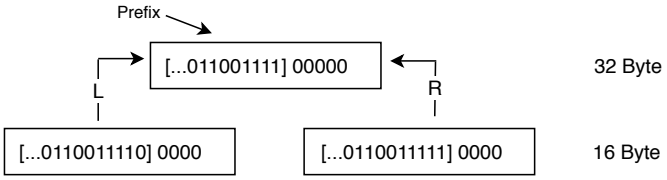
This is a small example of a single split, from a 32 byte block to two 16 byte blocks/buddies.



In this example, you can see that the prefix was effectively extended, with the LEFT block remaining the same (so that it corresponds to the address of the original block), and the RIGHT block turns on right-most bit. The original prefix digits remain the same.

COALESCING:

The coalescing process is very similar to the splitting process, but in reverse. When the time comes to merge the two 'buddy' blocks together, the process can only proceed if BOTH buddies are UNALLOCATED. Also, only two MATCHING blocks (same prefix-1, one L and one R) can coalesce. This ensures only original, contiguous blocks converge.



ETC:

Upsides: Simplifies coalescing, quick.
Downsides: Internal fragmentation occurs due to the 'rounding up' process.

Sources: OSTEP, chapter 17. Kenneth Knowlton's "A fast storage allocator", and Paul Wilson et. al "Dynamic Storage Allocation□: A Survey and Critical Review. Also Prof. Morrissey's lecture!

9.3 Buddy Allocator Example

For this example, assume that our system has 1 MB of memory that is managed with a buddy allocator. Allocation requests can be of any size up to 1 MB, but usually will be much less.

As can be seen, the memory "chunk" that will be allocated will be the power of 2 that is either equal to the request or the next above the requested amount. When coalescing, only correct adjacencies can be combined. That is, the proper left buddy must be paired with the proper right buddy.

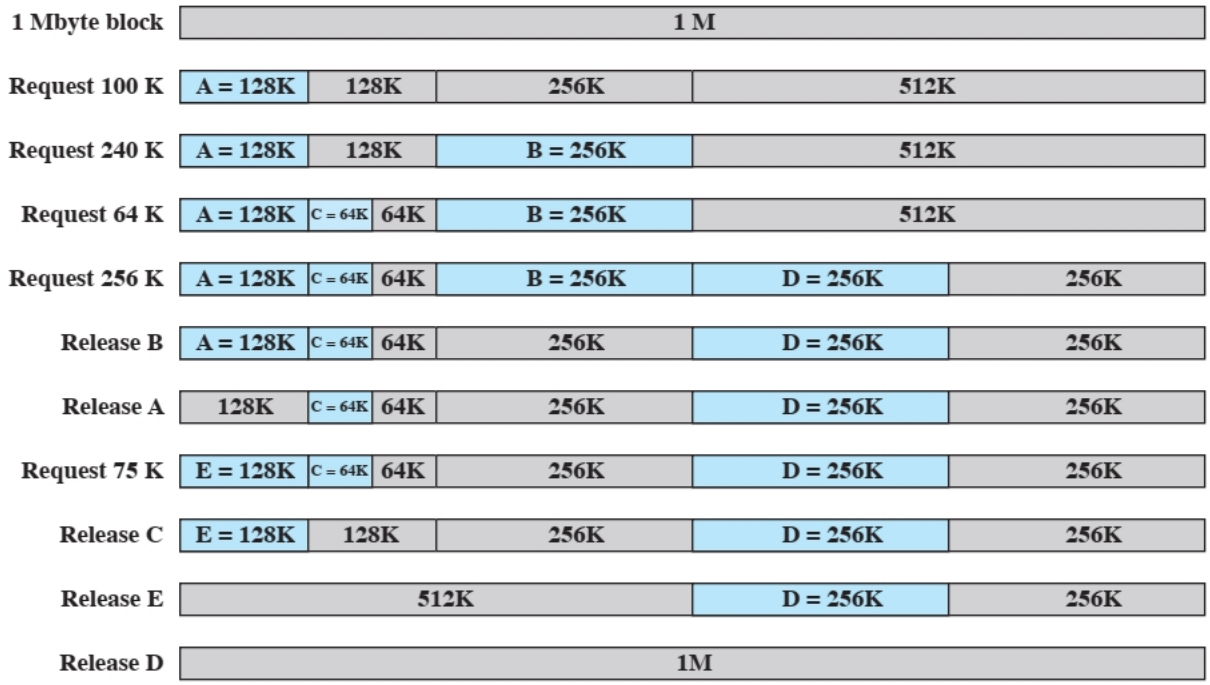


Figure 23: Example Buddy Allocator

Another way to look at this allocator is as a tree. Any one node may have one and only one parent. Leaf nodes represent nodes that can be allocated or have already been allocated; interior nodes cannot be allocated as they have already been split. Only two nodes, at the same level within the tree, that have the same parent node can coalesce, provided, of course, that they are both now free. Any node with a child cannot be involved in a coalescing step.

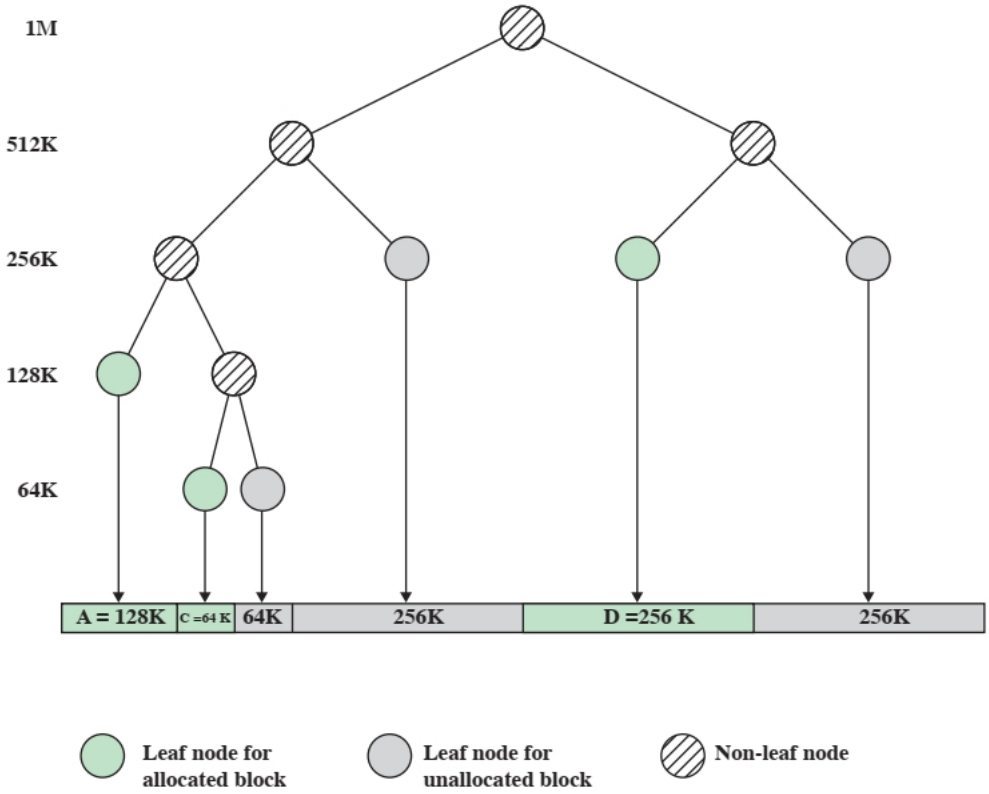


Figure 24: Buddy Allocator Tree Representation

These two figures show differing approaches to tracking free and in-use space. Depending on the system, either approach would be equally valid.

Figure 22 shows an interesting method for *coalescing* buddies. This simple approach is based on matching the *prefix* of the virtual address. At most two buddies will have the same prefix and telling the right from the left is trivial. A similar approach is used in networking where the technique is called *longest prefix match*. All we need do with this approach is to periodically scan a list. If there are two blocks with the same prefix, we know that they are free and thus can be coalesced. Pretty cool trick.

10 Paged Virtual Memory

We saw in segmentation that fixed sized segments have the benefit of no external fragmentation. But the problem of internal fragmentation still loomed large. We focused on internal fragmentation while acknowledging that variable sized segments opened the door to external fragmentation and a whole host of complications. The solutions all seemed to have undesirable features, such as memory compaction, during which no processes may run.

What if we could have the benefits of fixed segments and at the same time place a small, but acceptable bound on internal fragmentation. We hinted at this when we said that we could potentially have millions of segments. Of course, since the IA32 architecture only has 6 segment registers, moving to millions of smaller segments just increases the complexity of segment management for the programmer.

What if, instead we break our process into *pages*, which look like very small fixed sized segments, and then made it so that the programmer did not have to manage this new approach? The operating system and hardware would collaborate to manage memory use for all processes in the system.

The key ideas here are to partition virtual memory into *virtual pages* and physical memory into *page frames*. Pages can come in different sizes, but we will focus on the most common size.

10.1 Virtual Page

Virtual pages are typically 4KB in size. Not that this is 2^{12}, which means that we must steal 12 bit (0 though 11) to account for the offset into a page. If we are on a 32-bit system, this leaves 2^{20} bits, or 1MB worth of 4KB pages. This yields a virtual address space of 4GB. Quite a large memory space that the programmer doesn't have to manage in the same way as with segmentation.

So a virtual page is a fixed-length contiguous block of virtual memory, described by a single entry in what is called the page table. It is the smallest unit of allocation for memory in a virtual memory operating system. Similarly, a page frame is the smallest fixed-length contiguous block of physical memory into which virtual memory pages are mapped (loaded) by the operating system. Typically, the page frame is the same size as the virtual page, which is 4KB for our purposes.

A transfer of virtual pages between main memory and an auxiliary store, such as a hard disk drive, is referred to as paging or swapping. Swapping is a holdover from segmentation but is used in this context nonetheless.

10.2 Page Frame

A page frame holds a single virtual page, or is free. This means that there is a 1-1 correspondence between a virtual page in memory and the page frame that holds the page. We will soon see the concept of a *working set*, which is comprised of the virtual pages that are in active use by the process and thus need to be loaded into page frames. A process now looks like Figure 25. This approach, where only necessary virtual pages are loaded into page frames means that we can have more processes in main memory.

This is beneficial since the scheduler can only allocate a processor to a runnable process and in order to make meaningful progress, the working set must be in memory. Typically, the working set is a small fraction of the virtual pages in a process. In other words, paging allows more processes into main memory and therefor increases the likelihood that the scheduler will find a runnable process. Since a major goal of the operating system is efficient use of the processor(s), this is good.

10.3 Using the Disk

Conceptually, virtual memory is an array of N contiguous bytes stored on disk. Some contents of the array on disk are cached in physical memory. For a paged virtual memory system, the cache blocks are called pages, where a page size is $P = 2^p$ bytes. Physical memory can be viewed as the lowest level cache in our system[12].

The total amount of virtual memory in use may greatly exceed the amount of physical memory. To accommodate this, virtual memory also includes a section of disk space typically known as swap space[13]. Thus, the total amount of virtual memory available to a system at any point is the sum of physical memory and swap space[14].

What happens when a virtual page is required but is not currently in main memory?

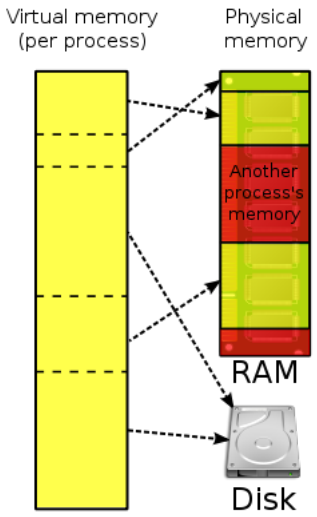


Figure 25: Virtual Memory with Paging

At any time, a page frame can hold one virtual page[15] for a process or be unused (free).

But what about fragmentation?

Herein lies the key benefit of paging with regards to memory usage. The virtual address space is based on the architecture word size, which is a power of 2. A virtual page is also a power of 2. We can see that there is *no external fragmentation!* Further, because the system allocates virtual pages on demand, the internal fragmentation is a function of page size.

Conceptually[16], the internal fragmentation will be less that one page; in other words, the maximum internal fragmentation is $2^{12} - 1$, basically 4KB, *no matter how large, or small, the process is!* Statistically, the average internal fragmentation will be about 2KB. You would be hard pressed to do this well with segmentation and paging comes with no overhead for the programmer. It is essentially free.

[12] We will not bring distributed or networked systems into the discussion at this time.

[13] Also called *backing store*.

[14] You should note that swap is not infinite. There will always be an upper bound on the amount of memory available to the system.

[15] There are systems wherein a page frame can hold more than one virtual page. Such systems require a more complex virtual address translation mechanism and so are beyond the scope of this class. We will only considered page frames that hold exactly one virtual page.

[16] It's more complicated but extrapolating from the example is straightforward.

10.4 Page Table

Every instruction fetch or data access can result in an access to physical memory[17]. The processor only *knows* about virtual addresses, assigned by the linker long before the process is created or dynamically created at run time.

Since we can't have millions of register for all our pages, we instead use a page table mechanism. The page table is stored in main memory and page table entries can be cached into a special cache inside the MMU, called the translation lookaside buffer, or TLB.

The page table has one entry for each possible virtual page[18]. Thus, the page table, in its simplest form, is just an array indexed by virtual page number.

The material in this section comes from the Intel 80386 Programmer's Reference Manual – Section 5.2.

The page table, generally stored in main memory, keeps track of where the virtual pages are stored in the physical memory. This method uses two memory accesses to access a single architecture word size of data. First, the page table is looked up for the frame number. Second, the frame number with the page offset gives the actual address. Thus any straightforward virtual memory scheme would have the effect of doubling the memory access time. Hence, the TLB is used to reduce the time taken to access the memory locations in the page table method. The TLB is a cache of the page table, representing only a subset of the page table contents.

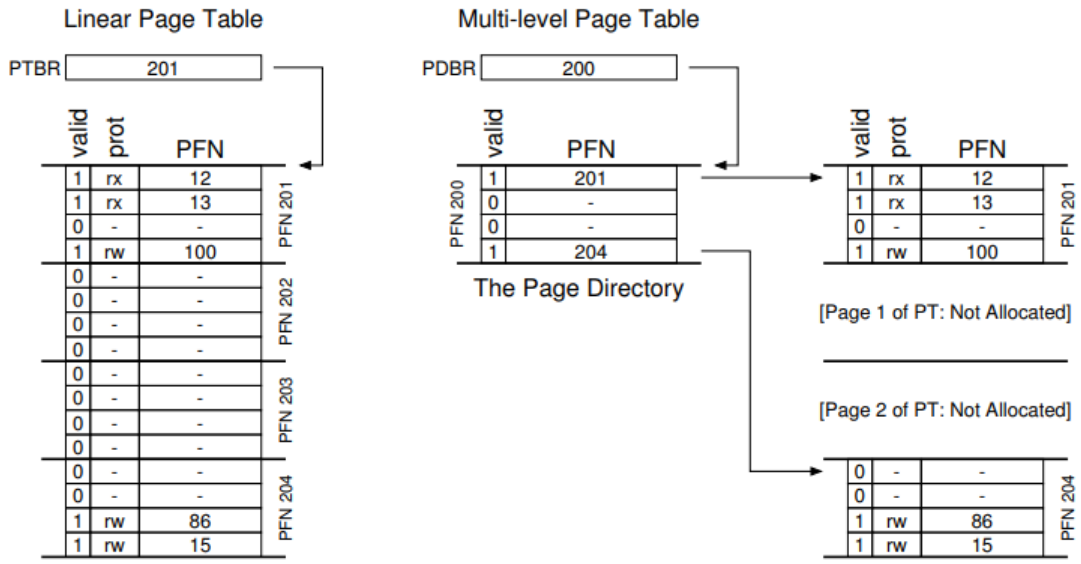


Figure 26: Page Tables

Note that, for IA32 and x86-64, the PTBR (Page Table Base Register) in Figure 26 is the CR3 control register.

[17] Don't forget about the role of memory caches.

[18] We'll work on fixing this later as it is grossly inefficient.

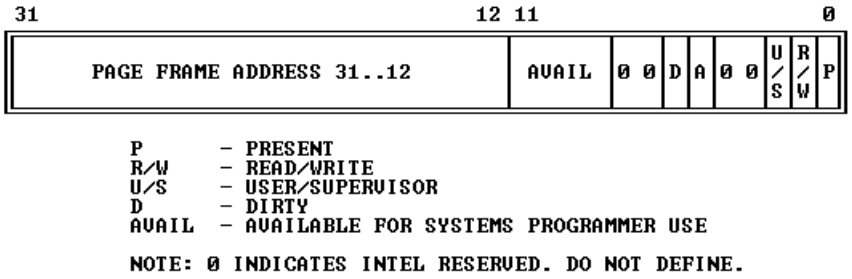


Figure 27: IA32 PTE Fields

Page Frame Address The page frame address specifies the physical starting address of a page. Because pages are located on 4K boundaries, the low-order 12 bits are always zero. In a page directory, the page frame address is the address of a page table.

Present Bit The Present bit indicates whether a page table entry can be used in address translation. P=1 indicates that the entry can be used. When P=0 in either level of page tables, the entry is not valid for address translation, and the rest of the entry is available for software use; none of the other bits in the entry is tested by the hardware.

If P=0 in either level of page tables when an attempt is made to use a page table entry for address translation, the processor signals a page exception. In software systems that support paged virtual memory, the page-not-present exception handler can bring the required page into physical memory.

Note that there is no present bit for the page directory itself. The page directory may be not-present while the associated task is suspended, but the operating system must ensure that the page directory indicated by the CR3 image in the TSS is present in physical memory before the task is dispatched .

Accessed and Dirty Bits These bits provide data about page usage in both levels of the page tables. With the exception of the dirty bit in a page directory entry, these bits are set by the hardware; however, the processor does not clear any of these bits.

The processor sets the corresponding accessed bits in both levels of page tables to one before a read or write operation to a page.

The processor sets the dirty bit in the second-level page table to one before a write to an address covered by that page table entry. The dirty bit in directory entries is undefined.

An operating system that supports paged virtual memory can use these bits to determine what pages to eliminate from physical memory when the demand for memory exceeds the physical memory available. The operating system is responsible for testing and clearing these bits.

Read/Write and User/Supervisor Bits These bits are not used for address translation, but are used for page-level protection, which the processor performs at the same time as address translation . Refer to Chapter 6 where protection is discussed in detail.

10.4.1 Page Hit

When a valid page frame holds the virtual page containing the requested virtual address, this is termed a hit.

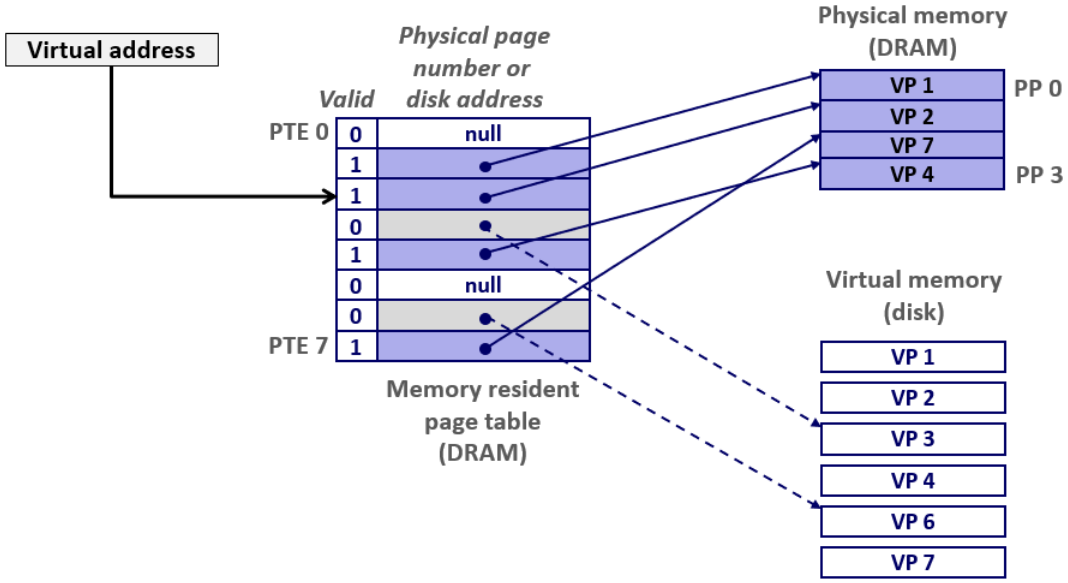


Figure 28: Page Hit

10.4.2 Page Fault

A page fault (miss) is a reference to virtual memory word that is not in physical memory. That is, the virtual page on which this word resides is not currently in a page frame. The fault is an exception that the operating system handles by getting the required virtual page from disk and placing it into a page frame. If there is not an unused page frame then the operating system must select a *victim page* to eject from its page frame in order to accommodate the faulting page.

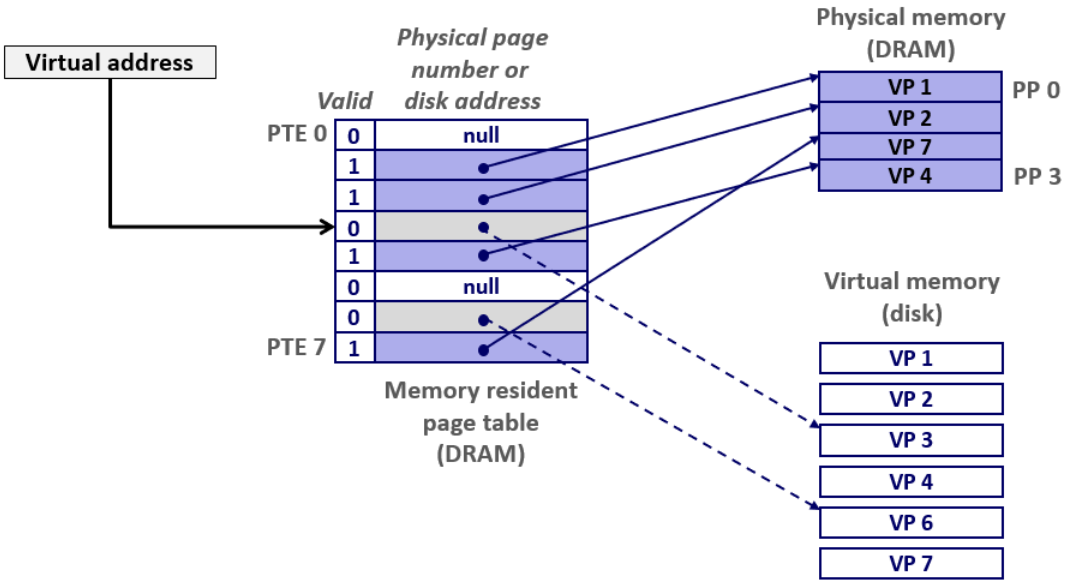


Figure 29: Page Fault

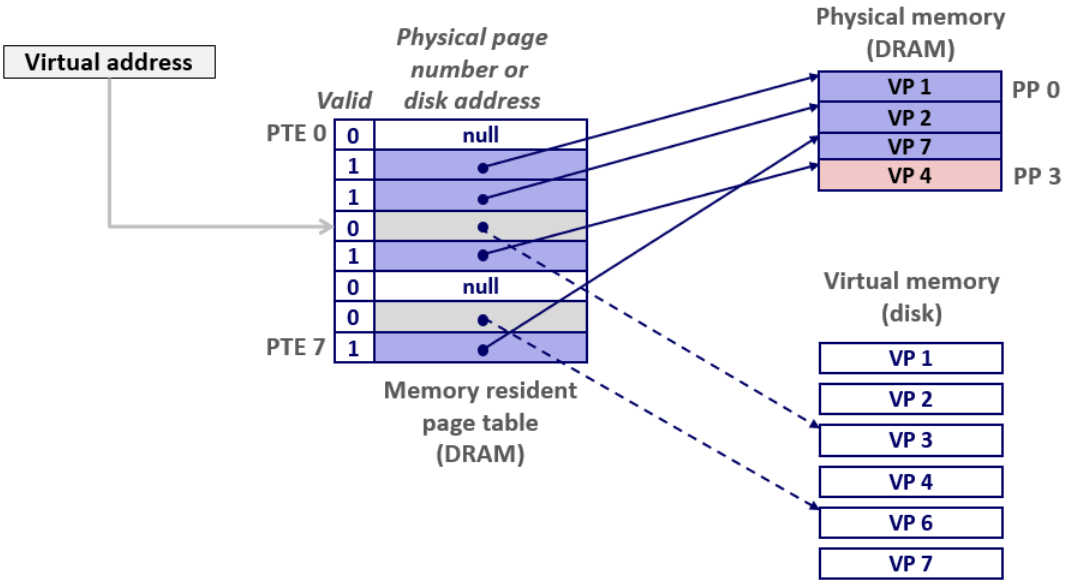


Figure 30: Selecting a Victim Page

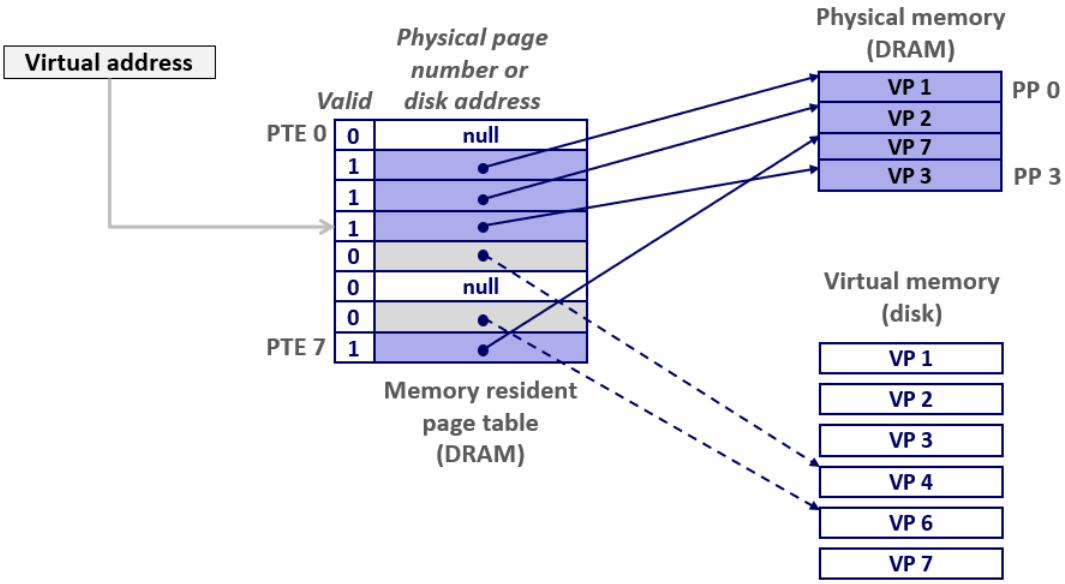


Figure 31: Replace Victim Page

Once the required page has been placed into a page frame, the faulting instruction can be *restarted*. Waiting until the miss to replace a page is known as *demand paging*. This is distinct from *prefetching* or other techniques.

10.5 Locality of Reference

The idea of locality of reference, also known as the principle of locality, is the tendency of a processor to access the same set of memory locations repetitively over a short period of time. There are two basic types of reference locality – temporal locality and spatial locality. Temporal locality refers to the reuse of specific data, and/or resources, within a relatively small time duration. Spatial locality refers to the use of data elements within relatively close storage locations. Sequential locality, a special case of spatial locality, occurs when data elements are arranged and accessed linearly, such as, traversing the elements in a one-dimensional array.

Locality is merely one type of predictable behavior that occurs in computer systems. Systems that exhibit strong locality of reference are great candidates for performance optimization through the use of techniques such as the caching, prefetching for memory and advanced branch predictors at the pipelining stage of a processor core[19].

[19] See Wikipedia.

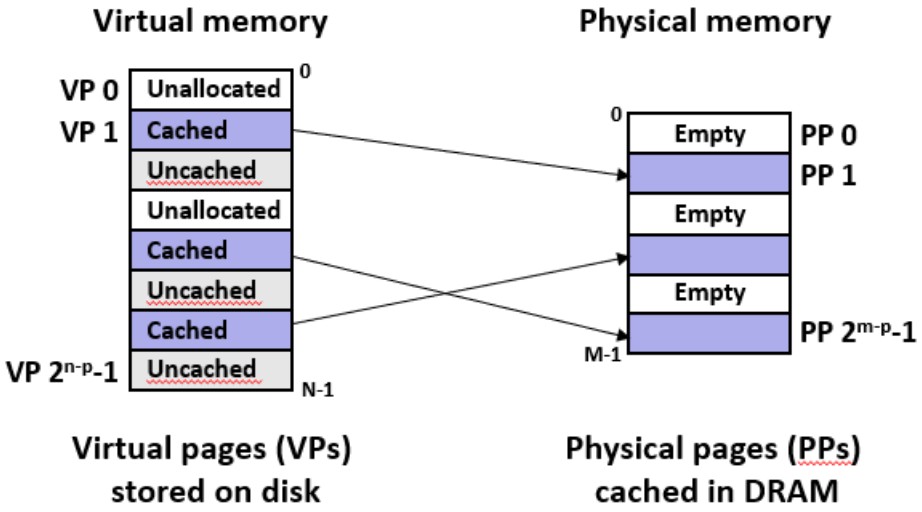


Figure 32: Locality of Reference

10.6 Working Set

In a paging system, the working set for any process is the set pages that comprises the current locality. Since future accesses by the process is impossible to predict, this approach uses a formula based on immediate past behavior by the process.

The choice of what pages to be kept in main memory (as distinct from being paged out to auxiliary storage) is important: if too many pages of a process are kept in main memory, then fewer other processes can be ready at any one time. If too few pages of a process are kept in main memory, then its page fault rate is greatly increased and the number of active (non-suspended) processes currently executing in the system approaches zero.

The working set model states that a process can be in RAM if and only if all of the pages that it is currently using (often approximated by the most recently used pages) can be in RAM. The model is an all or nothing model, meaning if the pages it needs to use increases, and there is no room in RAM, the process is swapped out of memory to free the memory for other processes to use.

Often a heavily loaded computer has so many processes queued up that, if all the processes were allowed to run for one scheduling time slice, they would refer to more pages than there is RAM, causing the computer to "thrash".

By swapping some processes from memory, the result is that processes – even processes that were temporarily removed from memory – finish much sooner than they would if the computer attempted to run them all at once. The processes also finish much sooner than they would if the computer only ran one process at a time to completion since it allows other processes to run and make progress during times that one process is waiting on the hard drive or some other global resource.

In other words, *the working set strategy* prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus it optimizes CPU utilization and throughput.

The *resident set size* (RSS) is the portion of memory occupied by a process that is held in main memory (RAM). The RSS must contain, at a minimum, the working set for the process. The rest

of the occupied memory exists in the swap space or file system, either because some parts of the occupied memory were paged out, or because some parts of the executable were never loaded.

10.7 Translation Lookaside Buffer

A translation lookaside buffer (TLB) is a memory cache that is used to reduce the time taken to access a user memory location. It is a part of the chip's memory-management unit (MMU). The TLB[20] stores the recent translations of virtual memory to physical memory and can be called an address-translation cache. A TLB may reside between the CPU and the CPU cache, between CPU cache and the main memory or between the different levels of the multi-level cache. The majority of desktop, laptop, and server processors include one or more TLBs in the memory-management hardware, and it is nearly always present in any processor that utilizes paged or segmented virtual memory.

The TLB is sometimes implemented as content-addressable memory (CAM). The CAM search key is the virtual address, and the search result is a physical address. If the requested address is present in the TLB, the CAM search yields a match quickly and the retrieved physical address can be used to access memory. This is called a TLB hit. If the requested address is not in the TLB, it is a miss, and the translation proceeds by looking up the page table in a process called a page walk. The page walk is time-consuming when compared to the processor speed, as it involves reading the contents of multiple memory locations and using them to compute the physical address. After the physical address is determined by the page walk, the virtual address to physical address mapping is entered into the TLB. The PowerPC 604, for example, has a two-way set-associative TLB for data loads and stores. Some processors have different instruction and data address TLBs.

A TLB has a fixed number of slots containing page table entries and segment-table entries; page table entries map virtual addresses to physical addresses and intermediate-table addresses, while segment-table entries map virtual addresses to segment addresses, intermediate-table addresses and page table addresses. The virtual memory is the memory space as seen from a process; this space is often split into pages of a fixed size (in paged memory), or less commonly into segments of variable sizes (in segmented memory). The page table, generally stored in main memory, keeps track of where the virtual pages are stored in the physical memory. This method uses two memory accesses (one for the page table entry, one for the byte) to access a byte. First, the page table is looked up for the frame number. Second, the frame number with the page offset gives the actual address. Thus any straightforward virtual memory scheme would have the effect of doubling the memory access time. Hence, the TLB is used to reduce the time taken to access the memory locations in the page table method. The TLB is a cache of the page table, representing only a subset of the page table contents.

Referencing the physical memory addresses, a TLB may reside between the CPU and the CPU cache, between the CPU cache and primary storage memory, or between levels of a multi-level cache. The placement determines whether the cache uses physical or virtual addressing. If the cache is virtually addressed, requests are sent directly from the CPU to the cache, and the TLB is accessed only on a cache miss. If the cache is physically addressed, the CPU does a TLB lookup on every memory operation, and the resulting physical address is sent to the cache.

In a Harvard architecture or modified Harvard architecture, a separate virtual address space or memory-access hardware may exist for instructions and data. This can lead to distinct TLBs for each access type, an instruction translation lookaside buffer (ITLB) and a data translation lookaside buffer (DTLB). Various benefits have been demonstrated with separate data and instruction

[20] See Wikipedia.

TLBs.

The TLB can be used as a fast lookup hardware cache. The figure shows the working of a TLB. Each entry in the TLB consists of two parts: a tag and a value. If the tag of the incoming virtual address matches the tag in the TLB, the corresponding value is returned. Since the TLB lookup is usually a part of the instruction pipeline, searches are fast and cause essentially no performance penalty. However, to be able to search within the instruction pipeline, the TLB has to be small.

A common optimization for physically addressed caches is to perform the TLB lookup in parallel with the cache access. Upon each virtual-memory reference, the hardware checks the TLB to see whether the page number is held therein. If yes, it is a TLB hit, and the translation is made. The frame number is returned and is used to access the memory. If the page number is not in the TLB, the page table must be checked. Depending on the CPU, this can be done automatically using a hardware or using an interrupt to the operating system. When the frame number is obtained, it can be used to access the memory. In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full, a suitable block must be selected for replacement. There are different replacement methods like least recently used (LRU), first in, first out (FIFO) etc.; see the address translation section in the cache article for more details about virtual addressing as it pertains to caches and TLBs.

Performance Implications

The flowchart in Figure 33 shows the working of a translation lookaside buffer. For simplicity, the page-fault routine is not mentioned. The CPU has to access main memory for an instruction-cache miss, data-cache miss, or TLB miss. The third case (the simplest one) is where the desired information itself actually is in a cache, but the information for virtual-to-physical translation is not in a TLB. These are all slow, due to the need to access a slower level of the memory hierarchy, so a well-functioning TLB is important. Indeed, a TLB miss can be more expensive than an instruction or data cache miss, due to the need for not just a load from main memory, but a page walk, requiring several memory accesses.

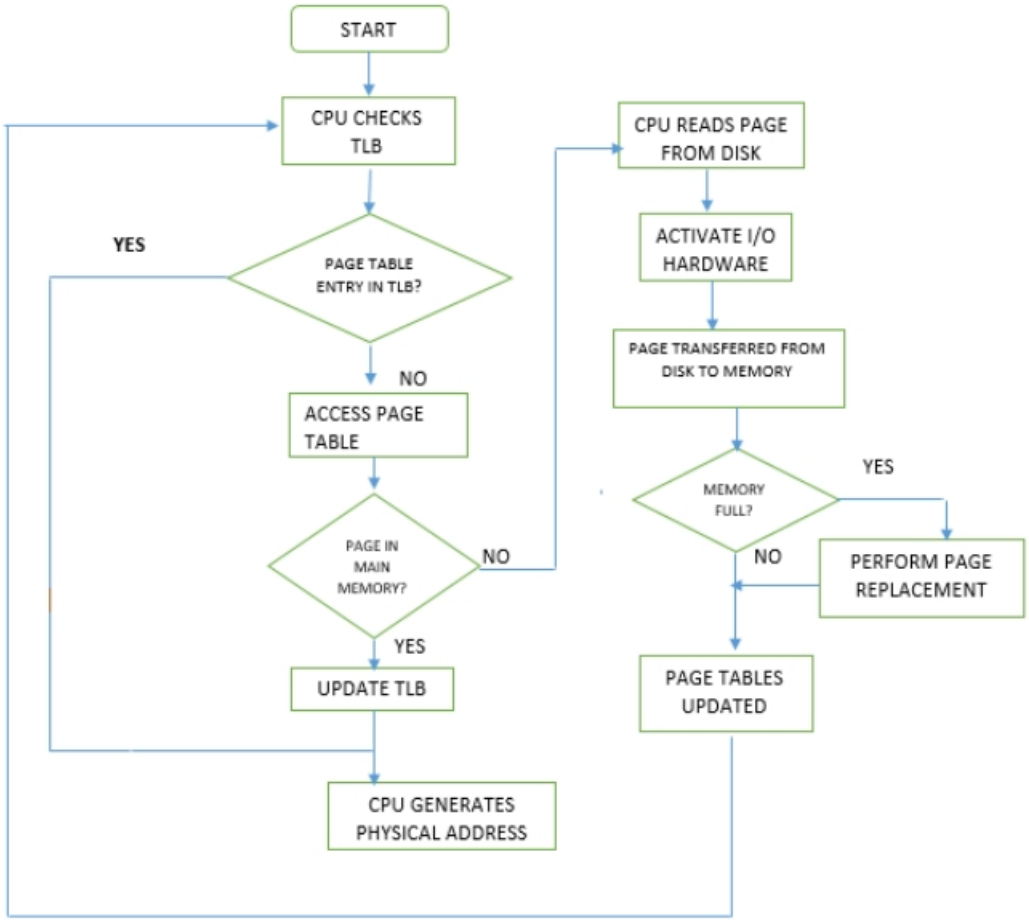


Figure 33: TLB Operation

The flowchart provided explains the working of a TLB. If it is a TLB miss, then the CPU checks the page table for the page table entry. If the present bit is set, then the page is in main memory, and the processor can retrieve the frame number from the page table entry to form the physical address. The processor also updates the TLB to include the new page table entry. Finally, if the present bit is not set, then the desired page is not in the main memory, and a page fault is issued. Then a page-fault interrupt is called, which executes the page-fault handling routine.

If the page working set does not fit into the TLB, then TLB thrashing occurs, where frequent TLB misses occur, with each newly cached page displacing one that will soon be used again, degrading performance in exactly the same way as thrashing of the instruction or data cache does. TLB thrashing can occur even if instruction-cache or data-cache thrashing are not occurring, because these are cached in different-size units. Instructions and data are cached in small blocks (cache lines), not entire pages, but address lookup is done at the page level. Thus even if the code and data working sets fit into cache, if the working sets are fragmented across many pages, the virtual-address working set may not fit into TLB, causing TLB thrashing. Appropriate sizing of the TLB thus requires considering not only the size of the corresponding instruction and data caches, but also how these are fragmented across multiple pages.

TLB Miss Handling

Two schemes for handling TLB misses are commonly found in modern architectures:

1. With hardware TLB management, the CPU automatically walks the page tables (using the CR3 register on x86, for instance) to see whether there is a valid page table entry for the specified virtual address. If an entry exists, it is brought into the TLB, and the TLB access is retried: this time the access will hit, and the program can proceed normally. If the CPU finds no valid entry for the virtual address in the page tables, it raises a page fault exception, which the operating system must handle. Handling page faults usually involves bringing the requested data into physical memory, setting up a page table entry to map the faulting virtual address to the correct physical address, and resuming the program. With a hardware-managed TLB, the format of the TLB entries is not visible to software and can change from CPU to CPU without causing loss of compatibility for the programs.

2. With software-managed TLBs, a TLB miss generates a TLB miss exception, and operating system code is responsible for walking the page tables and performing the translation in software. The operating system then loads the translation into the TLB and restarts the program from the instruction that caused the TLB miss. As with hardware TLB management, if the OS finds no valid translation in the page tables, a page fault has occurred, and the OS must handle it accordingly. Instruction sets of CPUs that have software-managed TLBs have instructions that allow loading entries into any slot in the TLB. The format of the TLB entry is defined as a part of the instruction set architecture (ISA). The MIPS architecture specifies a software-managed TLB; the SPARC V9 architecture allows an implementation of SPARC V9 to have no MMU, an MMU with a software-managed TLB, or an MMU with a hardware-managed TLB, and the UltraSPARC Architecture 2005 specifies a software-managed TLB.

The Itanium architecture provides an option of using either software- or hardware-managed TLBs.

Typical TLB Performance

Goal	Description
size	12 bits – 4,096 entries
hit time	0.5 – 1 clock cycle
miss penalty	10 – 100 clock cycles
miss rate	0.01 – 1% (20-40% for sparse/graph applications)

Table 6: TLB Performance

If a TLB hit takes 1 clock cycle, a miss takes 30 clock cycles, and the miss rate is 1%, the effective memory cycle rate is an average of 1 X 0.99 + (1 + 30) X 0.01 = 1.30 (1.30 clock cycles per memory access).

Address-space switch

On an address-space switch, as occurs on a process switch but not on a thread switch, some TLB entries can become invalid, since the virtual-to-physical mapping is different. The simplest strategy to deal with this is to completely flush the TLB. This means that after a switch, the TLB is empty, and any memory reference will be a miss, so it will be some time before things are running back at full speed. Newer CPUs use more effective strategies marking which process an

entry is for. This means that if a second process runs for only a short time and jumps back to a first process, it may still have valid entries, saving the time to reload them.

While selective flushing of the TLB is an option in software-managed TLBs, the only option in some hardware TLBs (for example, the TLB in the Intel 80386) is the complete flushing of the TLB on an address-space switch. Other hardware TLBs (for example, the TLB in the Intel 80486 and later x86 processors, and the TLB in ARM processors) allow the flushing of individual entries from the TLB indexed by virtual address.

Flushing of the TLB can be an important security mechanism for memory isolation between processes to ensure a process can't access data stored in memory pages of another process. Memory isolation is especially critical during switches between the privileged operating system kernel process and the user processes – as was highlighted by the Meltdown security vulnerability. Mitigation strategies such as kernel page table isolation (KPTI) rely heavily on performance-impacting TLB flushes and benefit greatly from hardware-enabled selective TLB entry management such as PCID.

The main problem with the Page Table is that it resides in main memory. This means that to access a single byte at a virtual address that two memory access are required: one to get the page translation information and one to get the data.

To speed address translation a hardware cache called a translation-lookaside buffer, or TLB is used. A TLB is part of the chip's memory-management unit (MMU), and is simply a hardware cache of popular virtual-to-physical address translations. Upon each virtual memory reference, the hardware first checks the TLB to see if the desired translation is present; if so, the translation is performed (quickly) without having to consult the page table (which has all translations). Because of their tremendous performance impact, TLBs in a real sense make virtual memory possible.

10.7.1 TLB Organization

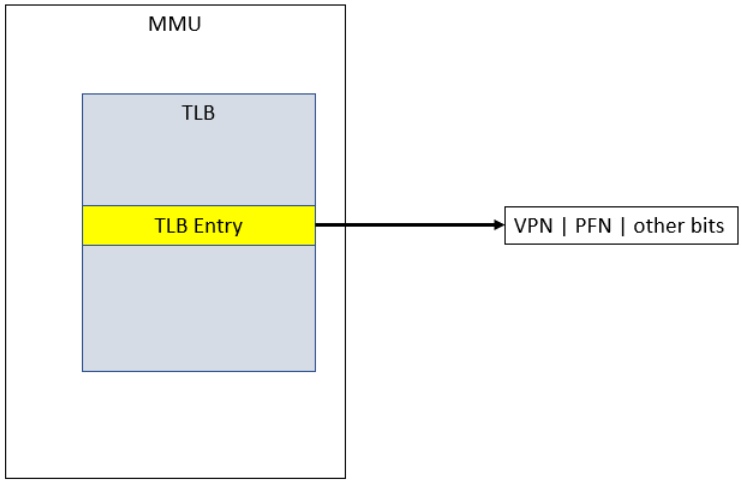


Figure 34: TLB Entry Format

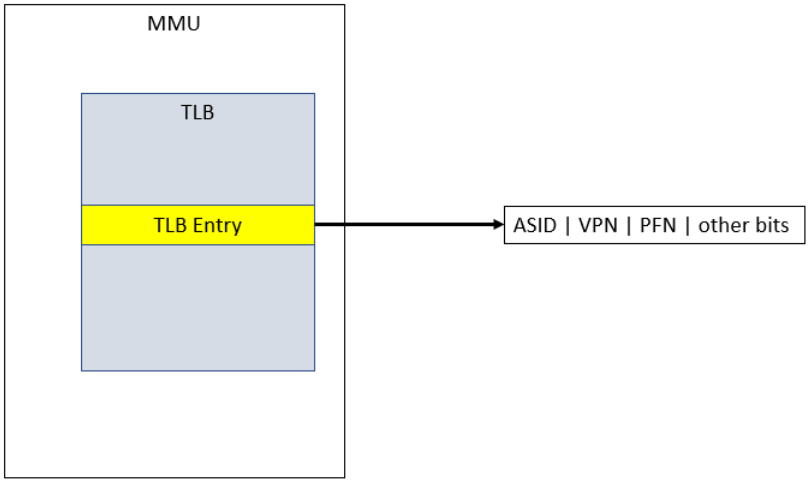


Figure 35: TLB Entry Format w/ ASID

More interesting are the "other bits". For example, the TLB commonly has a valid bit, which says whether the entry has a valid translation or not. Also common are protection bits, which determine how a page can be accessed (as in the page table). For example, code pages might be marked read and execute, whereas heap pages might be marked read and write. There may also be a few other fields, including an address-space identifier, a dirty bit, etc.

The TLB entry will have an indicator, typically a bit field, that contains, among other things, an indication as to whether the entry is *valid*. If the valid bit is not set, then the entry cannot be used. Clearing the valid bit on a TLB entry can happen, for example, when the page is paged out to disk and the page frame holding the translation is given to another page. The PTE will be marked to indicate that the entry is not present but any cached entries in all TLB entries for that page must be invalidated.

10.7.2 Context Switch Effect on TLB

On an address-space switch, as occurs on a process switch but not on a thread switch, some TLB entries can become invalid, since the virtual-to-physical mapping is different. The simplest strategy to deal with this is to completely flush the TLB. This means that after a switch, the TLB is empty (e.g., a *cold cache*), and any memory reference will be a miss, so it will be some time before things are running back at full speed. Newer CPUs use more effective strategies marking which process an entry is for. This means that if a second process runs for only a short time and jumps back to a first process, it may still have valid entries, saving the time to reload them.

One strategy is to use an *Address Space Identifier* (ASID), which functions similar to the process identifier, in that it uniquely associates one or more TLB entries with a single process address space. See Figure 35. The ASID increases the chance of a *warm cache* on context switch. Modern TLBs are quite large, meaning that, with the ASID approach, a warm cache is likely.

10.7.3 TLB Shootdown

A TLB is a cache of the translations from virtual memory addresses to physical memory addresses. In a multiprocessor or multicore system, when one processor changes the virtual-to-physical mapping of an address, the other processors need to invalidate that mapping if it is in their TLB. This

means that the next access to that TLB entry, for all but the processor initiating the shootdown, will pull the updated entry from the page table.
For example:

1. A page of virtual memory is shared among more than one processor at some time.
2. One processor updates the virtual-to-physical mapping, remapping to a new page frame or causing the virtual page to be swapped to disk.
3. All processor must be informed of this change or their TLB could contain a *stale mapping*, which would be *bad*.

The shootdown adds latency but it is preferable to the alternative of using an incorrect translation.

Students should consider the scenario where a virtual page is swapped to disk and the page frame reallocated to a new virtual page, possibly in a new process. Is there a potential for violation of the *isolation principle* and what could possibly be bad about that? Put another way, how does the TLB shootdown assist in providing *process isolation*?

10.7.4 Multi-Level TLB

10.8 Buffer Cache

Or, *How To Make I/O Seem Much Faster Than It Really Is.*

Disk access is slow. How could the disk latency be isolated so as not to impact process performance on read and write operations? In other words, how can we avoid disk accesses during page references by the process?

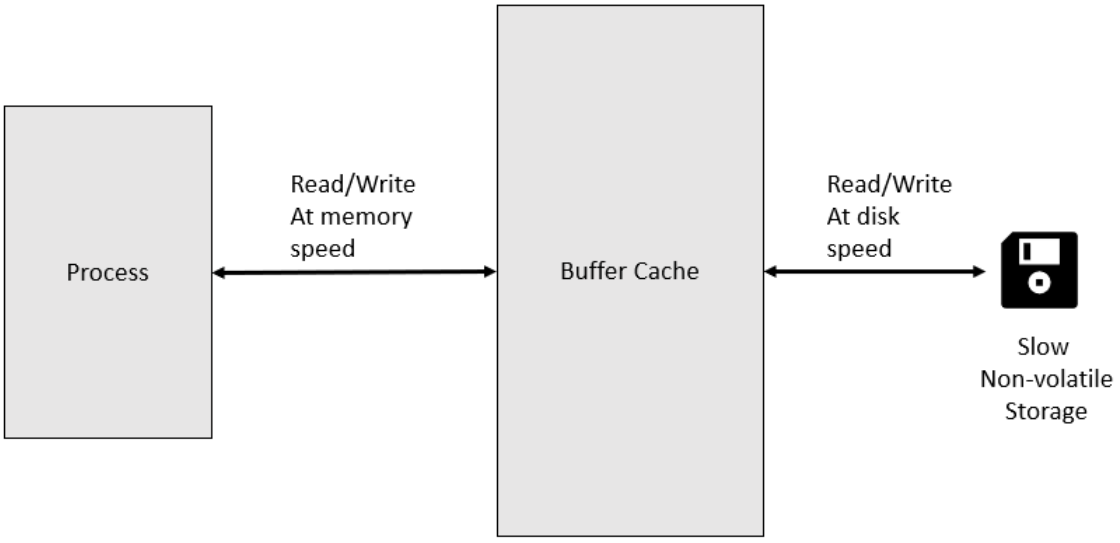


Figure 36: Buffer Cache

The buffer cache takes advantage of two interesting facts: modern disk drives are highly reliable and the power failures are rare[21]. What this implies is that the I/O subsystem can statistically get away with lying to the process.

If the buffer cache is holding the entire working set for your process, then reads and writes can be serviced by the buffer cache with only periodic writes to non-volatile storage for dirty pages. This causes the disk traffic to become *write dominated*, because all reads, except the first, to a page are handled by the buffer cache. Of course, the first time that a page is read, it will have to be read from disk.

Reads are now satisfied by the buffer cache. No need to access the disk for data one the page is loaded. This is true *even if the page has been modified*.

Writes are now "completed" (appears to complete) when they are written to the buffer cache. Thus, the process has much lower latency on write operations. The data can be flushed out to disk asynchronously to process execution. A very powerful benefit of this approach is write aggregation where multiple writes to a page are collected, or aggregated, in memory and then the page can be written once to disk. Aggregation thus turns many writes to a page into a single write of the page to disk.

How often do we flush changed pages to disk? Due to high reliability, systems typically set the flush interval to some value between 5 and 30 seconds. Think about how many writes could potentially be aggregated in this time period.

10.8.1 Algorithms

- Optimal
- FIFO
- LRU
- Clock

10.9 Page Replacement Algorithms

There are many more algorithms than shown here and we only show the most common[22].

10.9.1 FIFO

First-in first-out is simple to implement with small code and only occasionally very bad performance. The issue shows when the entire working set will not fit into memory.

Consider a scenario where the locality is comprised of m pages which are being accessed in a circular manner. Now assume that the process only has room for $m - 1$ pages in memory. Once the m^{th} page is accessed, all subsequent accesses will be to pages not in memory. In the general case, we only need a circular locality that is greater than the allowed number of pages in memory for FIFO to become a very poor choice.

Note that FIFO could wind up replacing a high-demand page merely because it has been in memory the longest.

[21] You do use an uninterruptible power supply (UPS) at home, right?

[22] See Wikipedia

10.9.2 LRU

For a given set of pages in memory, when we want to replace the one that will not be needed until the furthest time in the future. This optimal policy is infeasible since pages replacement policies are not omniscient. A reasonable approximation is to replace the page that has not been accessed for the longest period of time; that is, the least recently used page. This is an improvement over FIFO as it takes into account how often a page is used, not just when it was brought into memory.

A naïve implementation would have all pages on a list and every time a page is accessed, it will be moved to the front of the list. When a page needs to be replaced, the page at the back of the list is chosen. A simple list with head and tail pointers would suffice. List management overhead (latency) is a concern, so this approach is rarely used in practice.

10.9.3 Modified Clock

The clock algorithm is a more efficient implementation of LRU. The principle benefit is that there is no list management overhead. All pages are organized in a ring with a pointer to an arbitrary location. Conceptually, this looks like a clock with a single hand where the hand points to the page currently being evaluated for reuse.

We will call the approach described here "clock", even though we describe a clock variant. The original clock algorithm does not take into consideration whether or not a page is *dirty*. A dirty page is one that has been written to since it was last flushed to disk. If a page replacement algorithm wishes to reuse a dirty page, the page must first be written to disk; a high-latency activity.

The clock algorithm uses two bits: a *reference bit* and a *dirty bit*. The reference bit will determine if a page had been accessed recently (to approximate LRU) and the dirty page will determine if a page must be written to disk before reuse.

A key feature of the clock algorithm is the use of a *high watermark* and *low watermark* to manage background processes designed to help ensure that the likelihood of finding an unreferenced and *clean page* is high.

Let us assume that each page has associated with it a 2-tuple, (R,D), representing recently accessed (R) and dirty (D) conditions for pages. When selecting a page for replacement, (0,0) is preferred as there is no overhead to reusing the page. On the other extreme, a (1,1) page will have the highest overhead: it is likely to be referenced soon and would have to be written to disk (latency!) before it could be reused.

The algorithm is straightforward and can be done independent (asynchronously) of process activity (as a background activity).

- When a page is referenced, the R bit will be set.
- When data is written to a page, the D bit will be set.
- When a free page is requested, the following algorithm will be used.

  1. If the page pointed at is (0,0), use this page and advance the clock hand (pointer).
  2. If the page is (1,0) or (1,1), clear the reference bit and advance the pointer. Do not use this page.
  3. Asynchronously, dirty pages are flushed to disk. Optimally only the (0,1) pages are flushed since (1,1) pages most likely are involved in write aggregation. However, even (1,1) pages must eventually be flushed to persist the changes and guard against a system crash. Once a page has been successfully written to disk, its D bit is cleared.

It may seem that (0,1) pages cannot exist since any page that is written is also clearly accessed. However, a close look at the algorithm will show that (0,1) is not only valid, but likely; see step 2.

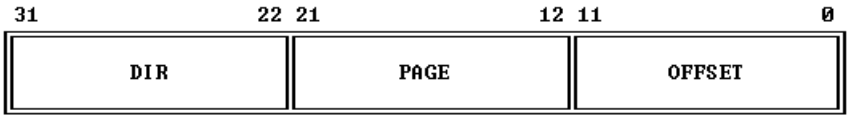
10.9.4 Related Issues

The OS also has to decide when to bring a page into memory. This policy, sometimes called the page selection policy, presents the OS with some different options.

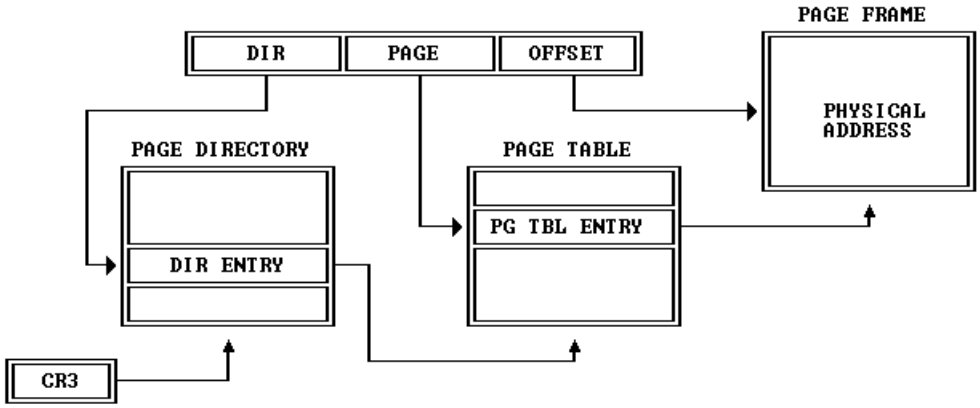
For most pages, the OS simply uses demand paging, which means the OS brings the page into memory when it is accessed, called "on demand". The OS could also guess that a page is about to be used, and bring it in ahead of time; this behavior is known as prefetching[23]. For example, some systems will assume that if a code page P is brought into memory, that code page $P+1$ will likely soon be accessed and thus should be brought into memory too.

Another policy determines how the OS writes pages out to disk. Of course, they could simply be written out one at a time; however, many systems instead collect a number of pending writes together in memory and write them to disk in one (more efficient) write. This behavior is usually called clustering or simply grouping of writes, and is effective because of the nature of disk drives, which perform a single large write more efficiently than many small ones.

10.10 Multi-Level Page Tables



(a) Intel 32-bit Address Format



(b) Translation with Multi-Level Page Tables

Figure 37: Intel 32-bit Address Interpretation

[23]Prefetching is a technique for "speeding up fetch operations" by beginning a fetch operation whose result is expected to be needed soon. Usually this is before it is known to be needed, so there is a risk of wasting time by prefetching data that will not be used. This technique really just reduces latency since the fetch of the page is started before an access occurs.

Intel supports an approach called **5-level paging** that does use many levels. Support was submitted as a set of patches to the Linux kernel on 8 December 2016.

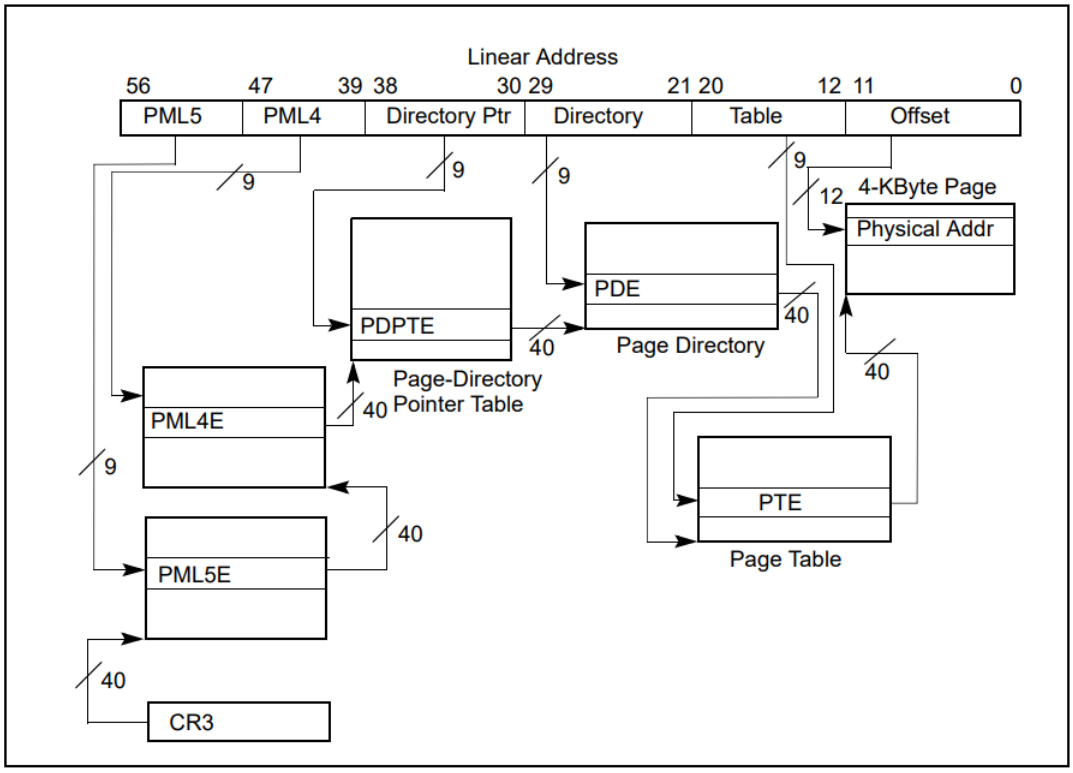


Figure 38: Intel 5-level Paging

10.11 Reverse Page Tables

Once the size of the architecture, which dictates the size of the virtual address space, gets to 64 bits, multi-level pages tables can be inefficient. One technique is to only track the page frames. This means that the OS must track which process and virtual page corresponds to each page frame. This can result in a large savings in memory overhead as opposed to creating page tables with many levels.

The Intel 5-level scheme is an alternative to reverse page tables.

11 Paged Virtual Memory – Putting It All Together

Add ASI to drawings

The idea is to use the "fast path" almost all the time. The idea is that the average translation latency should be as close to the fast path as possible. In these drawings, the fast path is comprised of steps $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$.

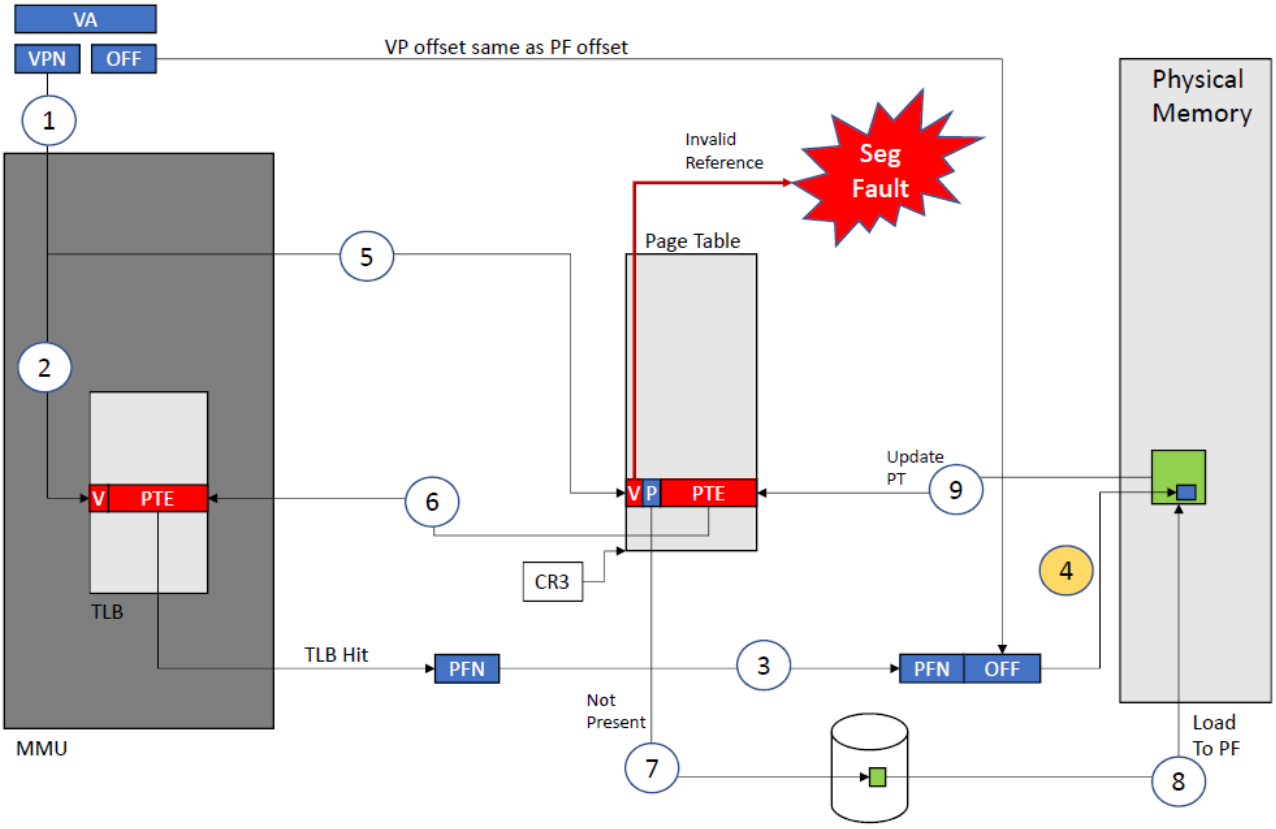


Figure 39: Single-level Page Table

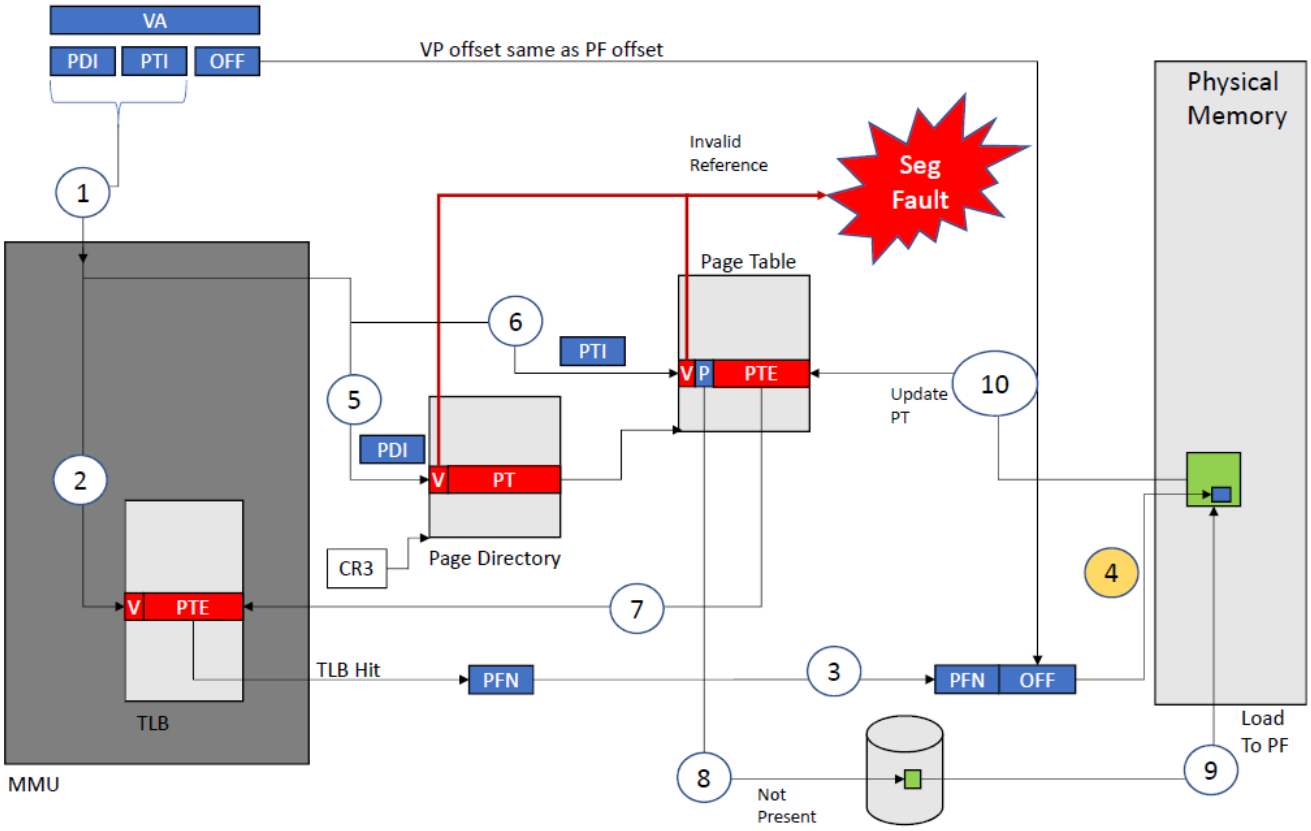


Figure 40: Multi-level Page Table

11.1 Additional Information

1. *Working Sets Past and Present*, P. J. Denning, in IEEE Transactions on Software Engineering, vol. SE-6, no. 1, pp. 64-84, Jan. 1980.
2. *The Working Set Model for Program Behavior*, Denning, Peter J, Communications of the ACM. 11. 323-333.
3. Additional papers and articles by Prof. Denning can be found at The Denning Institute.
4. *A Look at Several Memory Management Units, TLB-Refill Mechanisms, and Page Table Organizations*, Bruce L. Jacob and Trevor N. Mudge, e Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII), San Jose CA, Oct. 3-7, 1998.
5. *Hash, Don't Cache (the Page Table)*, Idan Yaniv and Dan Tsafrir, SIGMETRICS '16, June 14 - 18, 2016, Antibes Juan-Les-Pins, France.
6. *A Survey of Techniques for Architecting TLBs*, Sparsh Mittal, Indian Institute of Technology at Hyderabad.
7. *Translation Caching: Skip, Don't Walk (the Page Table)*, Thomas W. Barr, Alan L. Cox, and Scott Rixner. ISCA'10, June 19-23, 2010, Saint-Malo, France.

8. *Inter-Core Cooperative TLB Prefetchers for Chip Multiprocessors*, Abhishek Bhattacharjee and Margaret Martonosi, ASPLOS'10, March 13-17, 2010, Pittsburgh, Pennsylvania, USA.

12 Study Questions

1. Multiple choice: The translation lookaside buffer valid bit does which of the following? Select one:

   (a) Indicates if the VPN is mapped to a page frame that is currently in physical memory.
   (b) Indicates if the VPN is in an allocated part of the virtual address space for the process.
   (c) Indicates the cached PTE entry is correct for this address space.
   (d) Indicates the process has permission to write to this address in memory.

2. Multiple choice: The page table entry present bit does which of the following? Select one:

   (a) Indicates if the VPN is currently mapped to a valid page frame.
   (b) Indicates if the VPN is in an allocated part of the virtual address space for the process.
   (c) Indicates the cached PTE entry is correct for this address space.

3. Multiple choice: The page table entry valid bit does which of the following? Select one:

   (a) Indicates if the VPN is currently mapped to a valid page frame..
   (b) Indicates if the VPN is in an allocated part of the virtual address space for the process.
   (c) Indicates the cached PTE entry is correct for this process.

4. Multiple choice: When a page fault occurs which of the following will happen (select one):

   (a) Segmentation Violation
   (b) The operating system moves the requested page into a page frame in memory, and updates the page table.
   (c) The MMU moves the requested page into a page frame in memory, and updates the page table.
   (d) The operating system kills the process.
   (e) Double Free
   (f) The faulting instruction is skipped.

5. Multiple Choice: What is the result of trying to use a page directory index that is associated with a page directory entry for which the valid bit is not set? Select one:

   (a) Double Free
   (b) Page-Not-Present Exception
   (c) Segmentation Violation
   (d) The operating system moves the requested page into a page frame
   (e) The clock algorithm is invoked

6. The working set is (select one):

   (a) All of the memory a process will access while it is an active process.
   (b) All of the memory a process is currently using.

(c) All of the memory a process will use in the near future.

(d) All of the memory ever accessed by any process ever.

7. True/False: The total amount of virtual memory is always less than or equal to the amount of available physical memory.

8. True/False: An array allocated using a call to malloc() will be located in the stack.

9. Multiple Select: Select all of the following which are characteristics of the buddy allocation strategy.

(a) When a request from a process for memory is made to an empty list, a pointer to the head of the next larger list is returned to the process to fulfill its request.

(b) An array of lists are maintained pointing to free blocks of memory.

(c) The blocks of memory addresses are always aligned to a power of 2.

(d) Any two blocks on a list can be coalesced to a larger block for the next higher list.

(e) Any block on any list aside from the list with the smallest blocks can be split in half to populate the next lower list.

10. Multiple Choice: Compaction allows:

(a) Holes formed by variable-sized segmented allocation to be unified into a larger free block.

(b) Pages to be combined for larger allocations.

(c) The combining of different allocations from different levels of cache.

(d) Elimination of internal fragmentation.

11. Multiple Choice: Which of the following is not part of the Modified Clock LRU page replacement policy?

(a) Looks to find (0,0) page frame.

(b) If (0,0) allocate and move on.

(c) If (1,X) mark (0,X) and move on.

(d) If (0,1) send to device driver to write.

(e) If multiple (0,0) page frames, always choose the oldest.

12. Multiple Choice: What is the principle advantage of a multi-level page table over a single level page table?

(a) It improves space utilization in memory.

(b) It requires fewer accesses to main memory to obtain a PTE.

(c) Segmentation violations can be handled without killing the process.

(d) Compaction requires less time.

13. Multiple Choice: What is the principle advantage of a single level page table over a multi-level page table?

(a) It improves space utilization in memory.

(b) It requires fewer accesses to main memory to obtain a PTE.

(c) Segmentation violations can be handled without killing the process.

(d) Compaction requires less time.

14. Multiple Choice: Assuming a multi-level page table, which of the following is the correct interpretation of the present and valid bits in the page table entry (select one):

(a) Valid indicates that the page frame number is correct.
Present means that the page is in an allocated part of the address space.
If valid is not set, a segmentation fault results.

(b) Valid indicates the memory is working correctly.
Present indicates that there is enough memory in the system.
If both bits are 0, reboot.

(c) Valid indicates that VPN is in an allocated portion of
the process address space.
Present indicates that the referenced page is in memory. If the valid bit is 0, seg fault.

(d) Valid indicates that VPN is in an allocated portion of the process address space.
Present indicates that there is enough memory present.
If both bits are 0, the page is in memory.

(e) Valid indicates that the page table is valid.
Present indicates that the page table is present in memory.
If the valid bit is 0, seg fault.

(f) Valid indicates that the page is part of the process address space.
Present indicates that the page is in virtual memory.
If both are set, it means that the page is in the MMU.

(g) Valid indicates that the page is not part of the process address space.
Present indicates that the page is in virtual memory.
If both are set, it means that the page is in the MMU.

15. True/False: For a PTE, if the present bit is not set, the valid bit for the corresponding TLB entry will not be set.

16. True/False: The addition of an Address Space Identifier (ASID) to a TLB allows for the possibility of a warm cache on a context switch.

17. For segmentation and paging, choose an advantage and a disadvantage from the following list:

(a) No internal fragmentation.

(b) Allows efficient use of memory and eliminates external fragmentation.

(c) The buddy allocator is used for optimal memory allocation.

(d) Complex to implement.

(e) Requires a buffer cache.

(f) Simple to implement.

(g) Can waste memory unless compaction is used

18. In a single level page table, the virtual addresses consist of a Virtual Page Number (VPN) and an offset. What are the parts of a virtual address, in the correct order, when using a multi-level page table? Assume 2-levels.

    (a) Index Entry, Page Number, Offset
    (b) Page Table Index, Page Directory Index, Offset
    (c) Table of Contents, Chapter, Offset
    (d) Page Directory Entry, Page Table Entry, Offset
    (e) Page Directory Index, Page Table Index, Offset
    (f) Virtual Page Number, Page Frame Number, Offset

19. In a system that utilizes paging, what is the purpose of the Translation Lookaside Buffer (TLB)?

    (a) Garbage collection of unused memory.
    (b) Service calls to malloc and free
    (c) Cache mappings from virtual page numbers to page frame numbers to improve performance.
    (d) Translate physical addresses to virtual addresses.
    (e) Cache page table entries to improve performance.

20. In a system that utilizes paging, what is the purpose of the Memory Management Unit (MMU)?

    (a) Service calls to malloc and free
    (b) Translate virtual address offsets to physical address offsets
    (c) Translate physical addresses to virtual addresses
    (d) Manage physical memory.
    (e) Translate page frame numbers to virtual page numbers.
    (f) Translate virtual page numbers to page frame numbers.
    (g) Garbage collection of unused memory.

21. In a system with 16-bit virtual and physical addresses and single-level page tables with 2-byte entries, where the higher-order 7 bits of the address are the Virtual Page Number (VPN) and the lower-order 9 bits are the offset, what is the size of a page? Select one:

    (a) 4KB
    (b) 256 bytes
    (c) 512 bytes
    (d) 1KB

22. In a system with 16-bit virtual and physical addresses and single-level page tables with 2-byte entries, where the higher-order 7 bits of the address are the Virtual Page Number (VPN) and the lower-order 9 bits are the offset, how many entries are in a page table? Select one:

(a) 1024

(b) 64

(c) 128

(d) 256

23. In a system with 16-bit virtual and physical addresses and single-level page tables with 2-byte entries, where the higher-order 7 bits of the address are the Virtual Page Number (VPN) and the lower-order 9 bits are the offset, what is the size of a page table? Select one:

    (a) 4KB

    (b) 256 bytes

    (c) 512 bytes

    (d) 1KB

24. Suppose that TLB entries are structured as in the diagram. When will a process have a "cold" TLB cache?

    (a) At process creation

    (b) On a segmentation fault

    (c) When the present bit on any PTE is set

    (d) When there are no free page frames

    (e) On a context switch

25. A disadvantage of a cold TLB cache is

    (a) The working set for the current process is cached

    (b) Offset translation will take longer

    (c) The corresponding virtual page is not present in a page frame

    (d) The translation will have worst case latency if PTE present bit is not set

    (e) The clock algorithm will be unable to find a free page frame