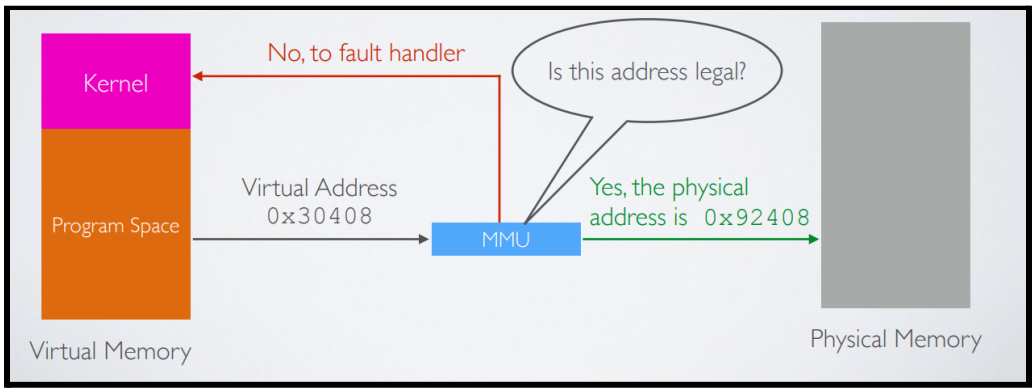
**Lecture Notes:**

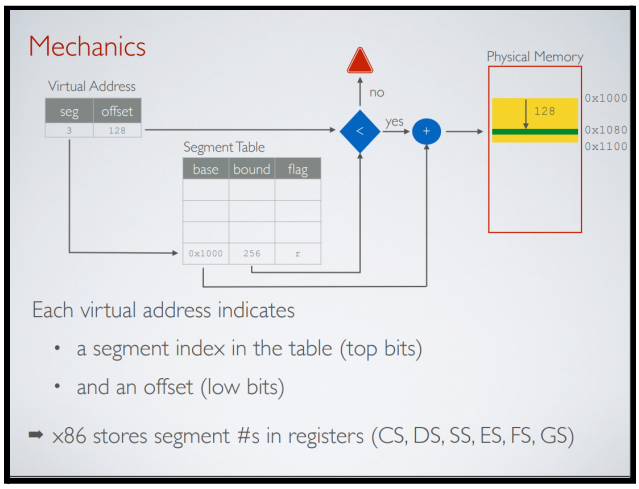
- **Virtual Memory:**
- While it is fine to place multiple execution contexts (stack and heap) at random locations in memory, having programs placed at random locations is problematic. Since function addresses and others are hard-encoded in the binary, the program cannot be placed at random locations in memory.
- A **compiler** takes source code files and translates (binds) symbolic addresses to logical, relocatable addresses within the compilation unit (object file).
- A **linker** takes a collection of object files and translates addresses to logical, absolute addresses within the executable (resolves references to symbols defined in other files/ modules).
- A naive approach is called **load time linking**. Here, we will do the linking and determine where the process will reside in memory and adjust all references within the program when the process is executed, not at compile time.
- However, this has a number of issues, such as:
    - How do we relocate the program in memory during execution? This is quite common because we're using functions and pointers.
    - What happens if there is no contiguous free region that fits the program?
    - How do we avoid programs interfering with each other?
- Issues with sharing physical memory include:
    1. **Transparency:**
    - A process shouldn't require particular physical memory bits.
    - A process often requires large amounts of contiguous memory (for stack, large data structures, etc).
    2. **Resource exhaustion:**
    - Programmers typically assume that the machine has enough memory but in reality, the sum of the sizes of all processes is often greater than physical memory.
    3. **Protection:**
    - How do we prevent A from observing B's memory?
    - How do we prevent process A from corrupting B's memory?
- We can use virtual memory to deal with these issues.
- Virtual memory goals:
    - Provide a convenient abstraction for programming by giving each program its own virtual address space.
    - Allow programs to see more memory than what exists.
    - Allocate scarce memory resources among competing processes to maximize performance with minimal overhead.
    - Enforce protection by preventing one process from messing with another's memory.
- Terminology:
    - Programs load/store to **virtual addresses**. Even the kernel.
    - Actual memory uses **physical addresses**.
    - Virtual memory hardware is called **MMU (Memory Management Unit)**. It is usually part of the CPU and is configured through privileged instructions. It translates from virtual to physical addresses and gives a per-process view of memory called the **address space**.
- The application does not see physical memory addresses. The MMU relocates each load/store at runtime. This means we can relocate processes while running either in memory or to disk.

I.e.

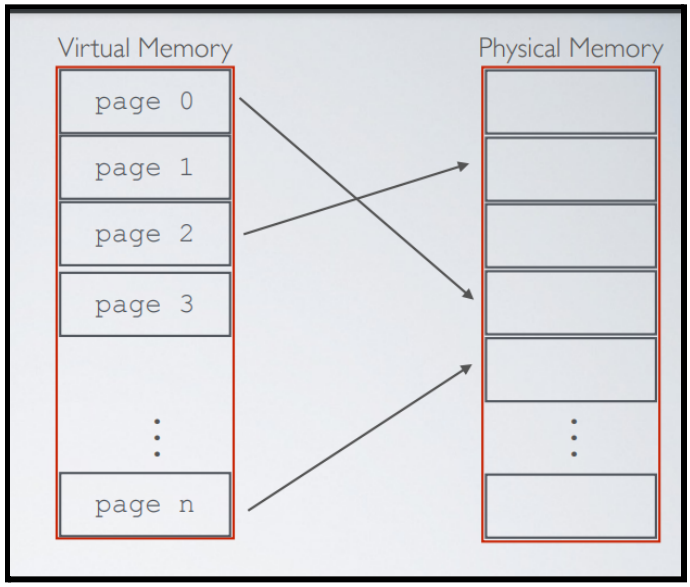



- Techniques for implementing virtual memory:
    1. Basic address translation
    2. Segmentation (the old way)
    3. Paging (the new way)
- **Basic Address Translation:**
- We have two special privileged registers: **base** and **bound**.
- On each load/store/jump:
    - First, set the physical address to be equal to virtual address + base
    - Then, check 0 ≤ virtual address < bound, else trap to kernel
- The OS can change these registers to move the process in memory.
- The OS must reload base for these registers on context switches. This is because each process has a different base value.
- Advantages:
    - Cheap in terms of hardware. There are only two registers.
    - Cheap in terms of cycles. Add and compare can be done in parallel.
- Disadvantages:
    - Growing a process is expensive.
    - No way to share code or data. One solution to this issue is **segmentation** (I.e. Have separate code, stack and data segments).
- **Segmentation:**
- The idea behind segmentation is that each process has a collection of multiple base/bound registers.
- This means that the address space is built from many segments (a.k.a segmentation table) and thus, it can share/protect memory at segment granularity.

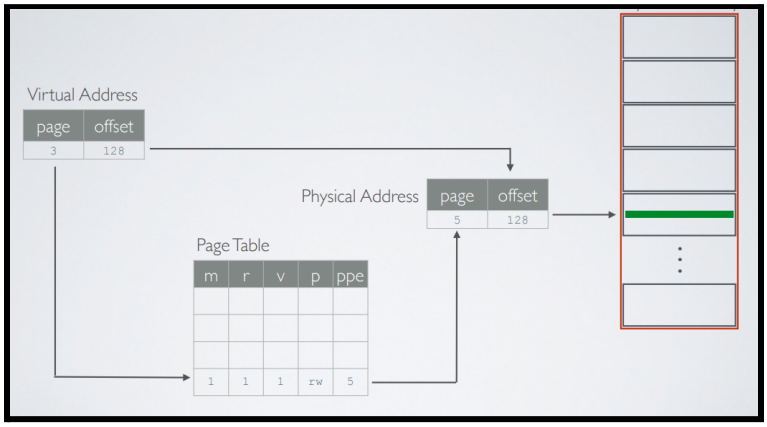
- The table works like this:




- Advantages:
    - Multiple segments per process (sparse memory).
    - Can easily share memory.
    - Do not need the entire process in memory (swap).
- Disadvantages:
    - Requires translation, which could limit performance.
    - Makes external **fragmentation** a real problem. Basically, you have a bunch of small chunks of free memory that can't be used.
- Fragmentation is the inability to use free memory.
- **External fragmentation** occurs because of variable sized pieces (Many small holes).
- **Internal fragmentation** occurs because of fixed size pieces (I.e. There are no external holes but an internal waste of space)
- **Paging:**
- The idea of paging is to divide the memory up into fixed-size pages, usually 4kb, to eliminate external fragmentation. Furthermore, each process has a collection of maps from virtual pages to physical pages. This can share/protect memory at page granularity. E.g.



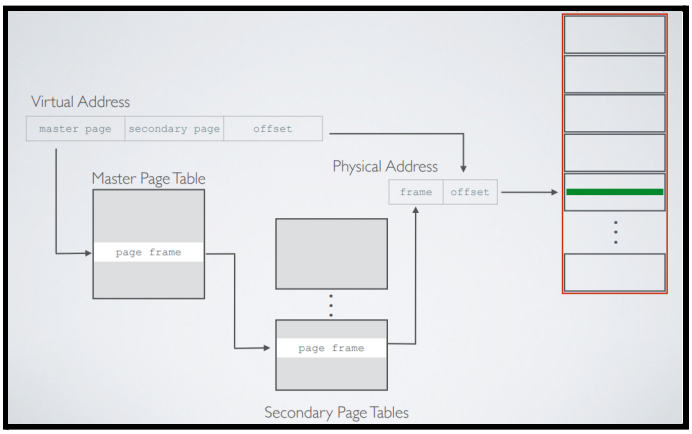
- Paging eliminates external fragmentation and simplifies allocation, free, and backing storage (swap). However, paging does cause internal fragmentation. The average internal fragmentation is .5 pages per "segment".
- Pages are fixed size (e.g. 4K) so a virtual address has two parts:
    1. **Virtual page number**, which is the most significant bits
    2. **Page offset**, which is the least significant 12 bits ($\log_2$ 4k)
- The page table is a collection of **page table entry (PTE)** that maps a **virtual page number (VPN)**, which is the index in the page table, to physical page numbers (PPN), which is also called frame number, and includes bits for protection, validity, etc.
- The **Modify bit** says whether or not the page has been written (set when the write to a page occurs).
- The **Reference bit** says whether the page has been accessed (set when a read or write to a page occurs).
- The **Valid bit** says whether or not the PTE can be used (checked each time the virtual address is used).
- The **Protection bits** say what operations (read, write, execute) are allowed on page.
- The **Physical page number (PPN)** determines the physical page.
- Page Lookup Example:




- Advantages:
    - Easy to allocate memory:
        - Memory comes from a free list of fixed size chunks.
        - Allocating a page is just removing it from the list.
        - External fragmentation is not a problem.
    - Easy to swap out chunks of a program:
        - All chunks are the same size.
        - Use a valid bit to detect references to swapped pages.
        - Pages are a convenient multiple of the disk block size.
- Disadvantages:
    - Can still have internal fragmentation.
    - Requires 2 or more references, which could limit performance.
      The solution is to use a hardware cache of lookups.
    - The amount of memory to store the page table is significant.
      Need one PTE per page, with 32 bit address space w/ 4KB pages = 2^20 PTEs.
      4 bytes/PTE = 4MB/page table
      25 processes = 100MB just for page tables
      The solution is to page the page tables.

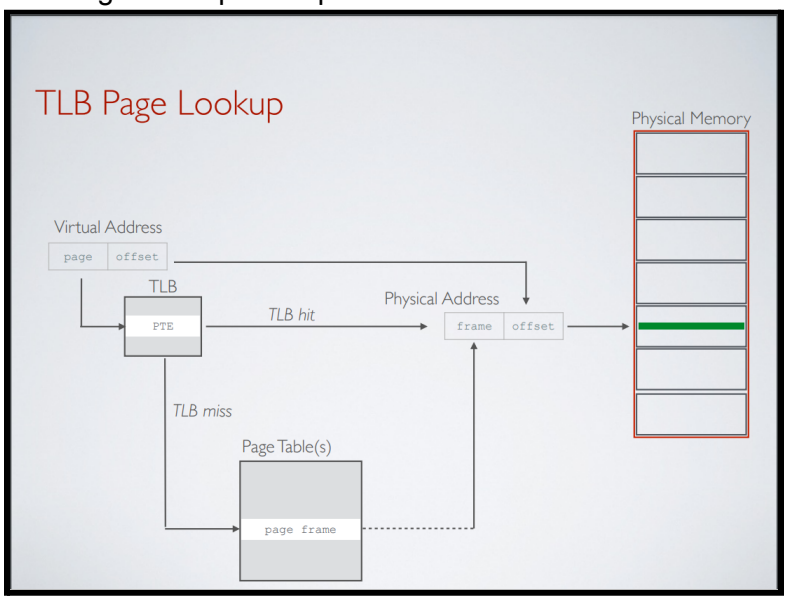
- **Improving Paging:**
    - Smaller page tables
    - Faster address translation
    - Larger virtual than physical memory (swapping)
    - Advanced Functionality
- **Smaller Page Tables:**
- Each process has a page table defining its address space.
- Considering 32-bit address space with 4K the size of the pages table is $2^{32} / 2^{12} \times 4B = 4MB$ / process this is a big overhead.
- The problem is that the page table is not part of the data structure and needs a lot of allocated memory. We need to make it sparse. The solution is that we only need to map the portion of the address space actually being used. Hence, we can use another level of indirection: **two-level page tables**.
- Virtual addresses in 2-level page tables have three parts:
    1. **Master page number**, which is the index in the master page table that maps to a secondary page table.
    2. **Secondary page number**, which is the index in the secondary page table that maps to the physical memory.
    3. **Offset** that indicates where the physical page address is located.

  I.e.




- **Faster Address Translation:**
- Translations take a lot of time, and with two-level page tables, we now have to do the translations twice.
- One-page table : one table lookup + one fetch
- Two-page table (32 bits) : 2 table lookups + one fetch
- 4-page table (64 bits) : 4 table lookup + one fetch
- A solution is to use a **Translation Lookaside Buffer (TLB)** which caches translations in hardware to reduce lookup cost.
- TLBs are special hardwares used to translate virtual page numbers into PTEs (not physical address) in a single machine cycle.
- It is typically 4-way to fully associative cache and all entries are looked up in parallel.
- It caches 32-128 PTE values (128-512K memory).
- TLBs exploit locality. Processes only use a handful of pages at a time so the TLB hit rate is very important for performances.
- 99% of the time, you hit the TLB and not the page table.

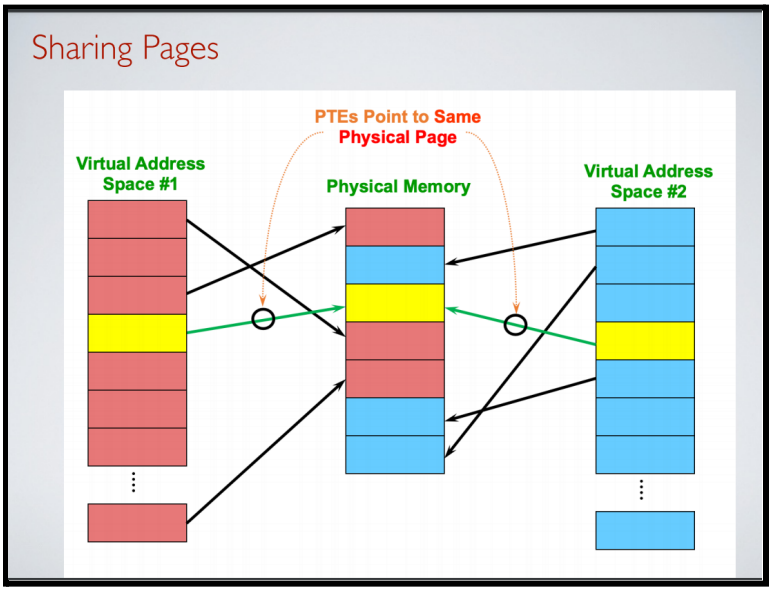
- TLB Page Lookup Example:




- Page lookup Steps:
  A process is executing on the CPU, and it issues a read to an address. The read goes to the TLB in the MMU.
  1. TLB does a lookup using the page number of the address.
  2. Common case is that the page number matches, returning a page table entry (PTE) for the mapping for this address.
  3. TLB validates that the PTE protection allows reads (in this example).
  4. PTE specifies which physical frame holds the page.
  5. MMU combines the physical frame and offset into a physical address.
  6. MMU then reads from that physical address, and returns value to the CPU.

  This is all done by the hardware.
- TLB misses can occur in 2 ways:
  1. TLB does not have a PTE mapping this virtual address.
  2. PTE in TLB, but memory access violates PTE protection bits.
- **Swapping:**
- The OS can use a disk to simulate a larger virtual memory than the physical memory meaning that pages can be moved between memory and disk (paging in/out).
- Paging process over time:
  - Initially, pages are allocated from memory.
  - When memory fills up, allocating a page requires some other page to be evicted.
  - Evicted pages go to disk, more precisely to the swap file/backing store.
  - Done by the OS, and transparent to the application.
- **Demand paging** is a method of virtual memory management. In a system that uses demand paging, the operating system copies a disk page into physical memory only if an attempt is made to access it and that page is not already in memory. It follows that a process begins execution with none of its pages in physical memory, and many page faults will occur until most of a process's working set of pages are located in physical memory. This is an example of a lazy loading technique.
- **Page Faults:**
- Read/write/execute protection bits are used to check and ensure some operations are not permitted on page. The TLB sends traps to the OS and the OS usually will send fault back up to the process.

- There are 2 possible reasons for invalid bits:
    1. Virtual page is not allocated:
        - The TLB sends traps to the OS and the OS sends a fault to the process. Then the software (OS) takes over.
    2. Virtual page is not allocated in the address space but is swapped on disk:
        - The TLB sends traps to the OS and the OS allocates a frame, reads from disk, and maps the PTE to a physical frame.
- Page fault steps:
    1. When the OS evicts a page, it sets the PTE as invalid and stores the location of the page in the swap file in the PTE.
    2. When a process accesses the page, the invalid PTE causes a trap (page fault).
    3. The trap will run the OS page fault handler.
    4. Handler uses the invalid PTE to locate the page in the swap file.
    5. Reads the page into a physical frame, updates PTE to point to it.
    6. Restarts the process.
- **Sharing:**
- Private VM spaces protect applications from each other but this makes it difficult to share data between processes. A solution is to have shared memory to allow processes to share data using direct memory references. This requires synchronization.
- E.g.




- Can we map shared memory at the same or different virtual addresses in each process' address space?
    - **Different Mapping:** Flexible but pointers inside shared memory are invalid.
    - **Same Mapping:** Less flexible but shared pointers are valid.

- **Copy on Write:**
- OSes spend a lot of time copying data.
  Here are 2 examples:
  1. System call arguments between user/kernel space.
  2. Entire address spaces to implement fork().
- Use **Copy on Write (CoW)** to defer large copies as long as possible, hoping to avoid them altogether.
  - Create shared mappings of parent pages in child virtual address space (instead of copying pages).
  - Shared pages are protected as read-only in parent and child.
    Any write operation will generate a protection fault, send a trap to the OS, copy the page, change page mapping in client page table, and restart write instruction.
- **Mapped Files:**
- **Mapped files** enable processes to do file I/O using loads and stores.
- Instead of "open, read into buffer, operate on buffer, etc" we bind a file to a virtual memory region.
  - PTEs map virtual addresses to physical frames holding file data.
  - Virtual address base + N refers to offset N in file.
- Initially, all pages mapped to file are invalid, similar to a swapped page.
  - The OS reads a page from a file when invalid page is accessed.
  - The OS writes a page to file when evicted, or region unmapped.
  - If the page is not dirty (has not been written to), no write is needed (another use of the dirty bit in PTE).
- Advantages:
  - Uniform access for files and memory (just use pointers).
- Disadvantages:
  - Process has less control over data movement as the OS handles faults transparently.
  - Does not generalize to streamed I/O (pipes, sockets, etc).
- **x86 architecture supports both paging and segmentation:**
- x86 architecture supports both paging and segmentation.

**Textbook Notes:**

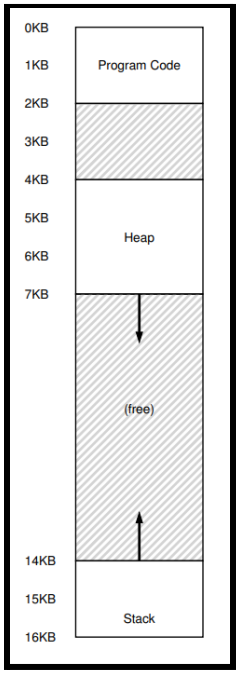
- **Mechanism - Address Translation:**
- Introduction:
- In developing the virtualization of the CPU, we focused on a general mechanism known as **limited direct execution (LDE)**.
- The idea behind LDE is simple. For the most part, let the program run directly on the hardware; however, at certain key points in time (such as when a process issues a system call, or a timer interrupt occurs), arrange so that the OS gets involved and makes sure the "right" thing happens. Thus, the OS, with a little hardware support, tries its best to get out of the way of the running program, to deliver an efficient virtualization. However, by interposing at those critical points in time, the OS ensures that it maintains control over the hardware. Efficiency and control together are two of the main goals of any modern operating system.
- **Interposition** is a generic and powerful technique that is often used to great effect in computer systems. In virtualizing memory, the hardware will interpose on each memory access, and translate each virtual address issued by the process to a physical address where the desired information is actually stored. However, the general technique of interposition is much more broadly applicable; indeed, almost any well-defined interface can be interposed upon, to add new functionality or improve some other aspect of the system. One of the usual benefits of such an approach is transparency; the interposition

often is done without changing the interface of the client, thus requiring no changes to said client.

- In virtualizing memory, we will pursue a similar strategy, attaining both efficiency and control while providing the desired virtualization. Efficiency dictates that we make use of hardware support, which at first will be quite rudimentary but will grow to be fairly complex.
- The generic technique we will use, which you can consider an addition to our general approach of limited direct execution, is something that is referred to as **hardware-based address translation/address translation**.
- With address translation, the hardware transforms each memory access such as 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. 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 memory.
- Of course, the hardware alone cannot virtualize memory, as it just provides the low-level mechanism for doing so efficiently. The OS must get involved at key points to set up the hardware so that the correct translations take place. It must thus manage memory, keeping track of which locations are free and which are in use, and judiciously intervening to maintain control over how memory is used.
- Assumptions:
- For now, we will assume these 3 things:
    1. The user's address space must be placed contiguously in physical memory.
    2. The size of the address space is not too big, specifically, that it is less than the size of physical memory.
    3. Each address space is exactly the same size.
- Dynamic (Hardware-based) Relocation:
- Also called **base and bounds**.
- For this implementation, we'll need two hardware registers within each CPU: one is called the **base register,** and the other the **bounds** (sometimes called a **limit register**).
- This base-and-bounds pair is going to allow us to place the address space anywhere we'd like in physical memory, and do so while ensuring that the process can only access its own address space.
- In this setup, each program is written and compiled as if it is loaded at address zero. However, when a program starts running, the OS decides where in physical memory it should be loaded and sets the base register to that value. Now, when any memory reference is generated by the process, it is translated by the processor in the following manner:
  physical address = virtual address + base
- Each memory reference generated by the process is a **virtual address**. The hardware in turn adds the contents of the **base register** to this address and the result is a **physical address** that can be issued to the memory system.
- With dynamic relocation, a little hardware goes a long way. Namely, a base register is used to transform virtual addresses (generated by the program) into physical addresses. A bounds or limit register ensures that such addresses are within the confines of the address space. Together they provide a simple and efficient virtualization of memory.
- Transforming a virtual address into a physical address is exactly the technique we refer to as **address translation**. This means the hardware takes a virtual address the process thinks it is referencing and transforms it into a physical address which is where the data actually resides. Because this relocation of the address happens at runtime, and because we can move address spaces even after the process has started running, the technique is often referred to as dynamic relocation.

- The bounds register is there to help with protection. Specifically, the processor will first check that the memory reference is within bounds to make sure it is legal.
- Bound registers can be defined in one of two ways:
    1. It holds the size of the address space, and thus the hardware checks the virtual address against it first before adding the base.
    2. It holds the physical address of the end of the address space, and thus the hardware first adds the base and then makes sure the address is within bounds.
- The CPU must be able to generate exceptions in situations where a user program tries to access memory illegally (with an address that is “out of bounds”). The OS handler can then figure out how to react, in this case likely terminating the process.
- The base and bounds registers are hardware structures kept on the chip (one pair per CPU). Sometimes people call the part of the processor that helps with address translation the **memory management unit (MMU)**.
- The OS must track which parts of free memory are not in use, so as to be able to allocate memory to processes. Many different data structures can of course be used for such a task. The simplest is a **free list**, which simply is a list of the ranges of the physical memory which are not currently in use.
- Operating System Issues:
- Just as the hardware provides new features to support dynamic relocation, the OS now has new issues it must handle; the combination of hardware support and OS management leads to the implementation of a simple virtual memory. Specifically, there are a few critical junctures where the OS must get involved to implement our base-and-bounds version of virtual memory.
- First, the OS must take action when a process is created, finding space for its address space in memory. Fortunately, given our assumptions that each address space is
  (a) smaller than the size of physical memory and
  (b) the same size, this is quite easy for the OS.
  it can simply view physical memory as an array of slots, and track whether each one is free or in use. When a new process is created, the OS will have to search through the free list to find room for the new address space and then mark it used.
- The OS must do some work when a process is terminated, specifically reclaiming all of its memory for use in other processes or the OS. Upon termination of a process, the OS thus puts its memory back on the free list, and cleans up any associated data structures as need be.
- The OS must also perform a few additional steps when a context switch occurs. There is only one base and bounds register pair on each CPU, after all, and their values differ for each running program, as each program is loaded at a different physical address in memory. Thus, the OS must save and restore the base-and-bounds pair when it switches between processes. Specifically, when the OS decides to stop running a process, it must save the values of the base and bounds registers to memory or sometimes in the process’s PCB. Similarly, when the OS resumes a running process or runs it the first time, it must set the values of the base and bounds on the CPU to the correct values for this process.
- We should note that when a process is stopped, it is possible for the OS to move an address space from one location in memory to another rather easily. To move a process’s address space, the OS first deschedules the process; then, the OS copies the address space from the current location to the new location; finally, the OS updates the saved base register in the process structure to point to the new location. When the process is resumed, its new base register is restored, and it begins running again, oblivious that its instructions and data are now in a completely new spot in memory.

- Lastly, the OS must provide exception handlers. The OS installs these handlers at boot time via privileged instructions.
- **Segmentation:**
- Segmentation - Generalized Base/Bounds:
- So far we have been putting the entire address space of each process in memory. With the base and bounds registers, the OS can easily relocate processes to different parts of physical memory. However, you might have noticed something interesting about these address spaces of ours: there is a big chunk of “free” space right in the middle, between the stack and the heap. Although the space between the stack and heap is not being used by the process, it is still taking up physical memory when we relocate the entire address space somewhere in physical memory; thus, the simple approach of using a base and bounds register pair to virtualize memory is wasteful. It also makes it quite hard to run a program when the entire address space doesn’t fit into memory; thus, base and bounds are not as flexible as we would like.
- I.e.




- To solve this problem, an idea was born, and it is called **segmentation**.
- The idea is simple: instead of having just one base and bounds pair in our MMU, why not have a base and bounds pair per logical segment of the address space? A segment is just a contiguous portion of the address space of a particular length, and in our canonical address space, we have three logically-different segments: code, stack, and heap. What segmentation allows the OS to do is to place each one of those segments in different parts of physical memory, and thus avoid filling physical memory with unused virtual address space.

- E.g. As you can see in the diagram, only used memory is allocated space in physical memory, and thus large address spaces with large amounts of unused address space, called **sparse address spaces**, can be accommodated.

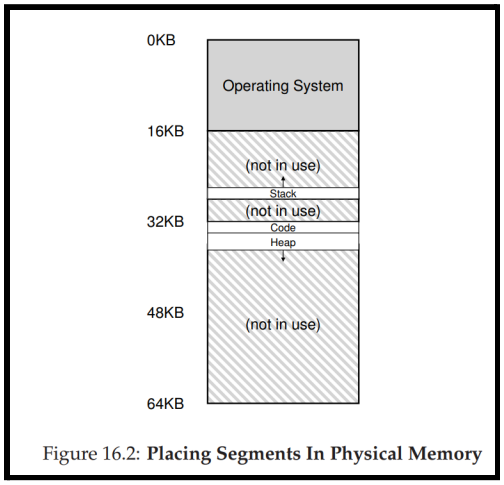


Figure 16.2: **Placing Segments In Physical Memory**

- The term **segmentation fault** arises from a memory access on a segmented machine to an illegal address.
- Which Segment Are We Referring To?:
- The hardware uses segment registers during translation, but how does it know the offset into a segment, and to which segment an address refers?
- One common approach, sometimes referred to as an **explicit approach**, is to chop up the address space into segments based on the top few bits of the virtual address.
- There are other ways for the hardware to determine which segment a particular address is in. In the **implicit approach**, the hardware determines the segment by noticing how the address was formed.
- Support for Sharing:
- As support for segmentation grew, system designers soon realized that they could realize new types of efficiencies with a little more hardware support. Specifically, to save memory, sometimes it is useful to share certain memory segments between address spaces. **Code sharing** is common and still in use in systems today.
- To support sharing, we need a little extra support from the hardware, in the form of **protection bits**.
- Basic support adds a few bits per segment, indicating whether or not a program can read or write a segment, or perhaps execute code that lies within the segment. By setting a code segment to read-only, the same code can be shared across multiple processes, without worry of harming isolation; while each process still thinks that it is accessing its own private memory, the OS is secretly sharing memory which cannot be modified by the process, and thus the illusion is preserved.
- Fine-grained vs. Coarse-grained Segmentation:
- Most of our examples thus far have focused on systems with just a few segments ( code, stack, heap). We can think of this segmentation as **coarse-grained**, as it chops up the address space into relatively large, coarse chunks. However, some early systems were more flexible and allowed for address spaces to consist of a large number of smaller segments, referred to as **fine-grained segmentation**.
- Supporting many segments requires even further hardware support, with a **segment table** of some kind stored in memory. Such segment tables usually support the creation of a very large number of segments, and thus enable a system to use segments in more flexible ways.

- OS Support:
- You now should have a basic idea as to how segmentation works. Pieces of the address space are relocated into physical memory as the system runs, and thus a huge savings of physical memory is achieved relative to our simpler approach with just a single base/bounds pair for the entire address space. Specifically, all the unused space between the stack and the heap need not be allocated in physical memory, allowing us to fit more address spaces into physical memory and support a large and sparse virtual address space per process.
- A general problem that arises is that physical memory quickly becomes full of little holes of free space, making it difficult to allocate new segments, or to grow existing ones. We call this problem **external fragmentation**.
- One solution to this problem would be to **compact physical memory** by rearranging the existing segments. For example, the OS could stop whichever processes are running, copy their data to one contiguous region of memory, change their segment register values to point to the new physical locations, and thus have a large free extent of memory with which to work. By doing so, the OS enables the new allocation request to succeed. However, compaction is expensive, as copying segments is memory-intensive and generally uses a fair amount of processor time.
- **Paging - Introduction:**
- A Simple Example And Overview:
- It is sometimes said that the operating system takes one of two approaches when solving most any space-management problem. The first approach is to chop things up into variable-sized pieces, as we saw with **segmentation** in virtual memory. Unfortunately, this solution has inherent difficulties. In particular, when dividing a space into different-size chunks, the space itself can become fragmented, and thus allocation becomes more challenging over time. Thus, it may be worth considering the second approach: to chop up space into fixed-sized pieces. In virtual memory, we call this idea **paging**.
- Instead of splitting up a process's address space into some number of variable-sized logical segments (code, heap, stack), we divide it into fixed-sized units, each of which we call a **page**. Correspondingly, we view physical memory as an array of fixed-sized slots called **page frames**. Each of these frames can contain a single virtual-memory page.
- The most important improvement of paging will be flexibility. With a fully-developed paging approach, the system will be able to support the abstraction of an address space effectively, regardless of how a process uses the address space.
- Another advantage is the simplicity of free-space management that paging affords.
- To record where each virtual page of the address space is placed in physical memory, the operating system usually keeps a per-process data structure known as a **page table**. The major role of the page table is to store address translations for each of the virtual pages of the address space, thus letting us know where in physical memory each page resides.
- Where Are Page Tables Stored?:
- Page tables can get terribly large, much bigger than the small segment table or base/bounds pair we have discussed previously.
- Because page tables are so big, we don't keep any special on-chip hardware in the MMU to store the page table of the currently-running process. Instead, we store the page table for each process in memory.
- What's Actually In The Page Table?:
- The page table is just a data structure that is used to map virtual addresses (or virtual page numbers) to physical addresses (physical frame numbers).

- The simplest form is called a **linear page table**, which is just an array. The OS indexes the array by the virtual page number (VPN), and looks up the page-table entry (PTE) at that index in order to find the desired physical frame number (PFN).
- As for the contents of each PTE, we have a number of different bits in there worth understanding at some level.
- A **valid bit** is common to indicate whether the particular translation is valid; for example, when a program starts running, it will have code and heap at one end of its address space, and the stack at the other. All the unused space in-between will be marked invalid, and if the process tries to access such memory, it will generate a trap to the OS which will likely terminate the process. Thus, the valid bit is crucial for supporting a sparse address space; by simply marking all the unused pages in the address space invalid, we remove the need to allocate physical frames for those pages and thus save a great deal of memory.
- We also might have **protection bits**, indicating whether the page could be read from, written to, or executed from. Again, accessing a page in a way not allowed by these bits will generate a trap to the OS.
- A **present bit** indicates whether this page is in physical memory or on disk.
- A **dirty bit** is also common, indicating whether the page has been modified since it was brought into memory.
- A **reference bit/accessed bit** is sometimes used to track whether a page has been accessed, and is useful in determining which pages are popular and thus should be kept in memory.
- Paging - Also Too Slow:
- With page tables in memory, we already know that they might be too big. As it turns out, they can slow things down too.
- **Paging - Faster Translations (TLBs):**
- TLB Basic Algorithm:
- Using paging as the core mechanism to support virtual memory can lead to high performance overheads. By chopping the address space into small, fixed-sized units (pages), paging requires a large amount of mapping information. Because that mapping information is generally stored in physical memory, paging logically requires an extra memory lookup for each virtual address generated by the program. Going to memory for translation information before every instruction fetch or explicit load or store is prohibitively slow.
- When we want to make things fast, the OS usually needs some help. And help often comes from the OS's old friend: the hardware. To speed address translation, we are going to add what is called a **translation-lookaside buffer** or **TLB**.
- 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 held therein. If so, the translation is performed quickly without having to consult the page table. Because of their tremendous performance impact, TLBs in a real sense make virtual memory possible.

- Here is a basic algorithm for TLBs:

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)    // TLB Hit
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset   = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = AccessMemory(PhysAddr)
8      else
9          RaiseException(PROTECTION_FAULT)
10 else                    // TLB Miss
11     PTEAddr = PTBR + (VPN * sizeof(PTE))
12     PTE = AccessMemory(PTEAddr)
13     if (PTE.Valid == False)
14         RaiseException(SEGMENTATION_FAULT)
15     else if (CanAccess(PTE.ProtectBits) == False)
16         RaiseException(PROTECTION_FAULT)
17     else
18         TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
19         RetryInstruction()
```

Figure 19.1: **TLB Control Flow Algorithm**

The algorithm the hardware follows works like this: first, extract the virtual page number (VPN) from the virtual address (Line 1), and check if the TLB holds the translation for this VPN (Line 2). If it does, we have a **TLB hit**, which means the TLB holds the translation. We can now extract the page frame number (PFN) from the relevant TLB entry, concatenate that onto the offset from the original virtual address, and form the desired physical address (PA), and access memory (Lines 5–7), assuming protection checks do not fail (Line 4). If the CPU does not find the translation in the TLB (a **TLB miss**), we have some more work to do. In this example, the hardware accesses the page table to find the translation (Lines 11–12), and, assuming that the virtual memory reference generated by the process is valid and accessible (Lines 13, 15), updates the TLB with the translation (Line 18). These sets of actions are costly, primarily because of the extra memory reference needed to access the page table (Line 12). Finally, once the TLB is updated, the hardware retries the instruction; this time, the translation is found in the TLB, and the memory reference is processed quickly.

- The TLB, like all caches, is built on the premise that in the common case, translations are found in the cache. If so, little overhead is added, as the TLB is found near the processing core and is designed to be quite fast. When a miss occurs, the high cost of paging is incurred; the page table must be accessed to find the translation, and an extra memory reference (or more, with more complex page tables) results. If this happens often, the program will likely run noticeably more slowly; memory accesses, relative to most CPU instructions, are quite costly, and TLB misses lead to more memory accesses. Thus, it is our hope to avoid TLB misses as much as we can.
- Caching is one of the most fundamental performance techniques in computer systems. The idea behind hardware caches is to take advantage of locality in instruction and data references.
- There are usually two types of locality: **temporal locality** and **spatial locality**. With temporal locality, the idea is that an instruction or data item that has been recently accessed will likely be re-accessed soon in the future. Think of loop variables or instructions in a loop; they are accessed repeatedly over time. With spatial locality, the idea is that if a program accesses memory at address x, it will likely soon access memory near x. Imagine here streaming through an array of some kind, accessing one element and then the next. Of course, these properties depend on the exact nature of the program, and thus are not hard-and-fast laws but more like rules of thumb.
- Hardware caches, whether for instructions, data, or address translations take advantage of locality by keeping copies of memory in small, fast on-chip memory. Instead of having to go to memory to satisfy a request, the processor can first check if a nearby copy exists in a cache; if it does, the processor can access it quickly and avoid spending the

costly time it takes to access memory. You might be wondering: if caches are so great, why don't we just make bigger caches and keep all of our data in them? Unfortunately, this is where we run into more fundamental laws like those of physics. If you want a fast cache, it has to be small, as issues like the speed-of-light and other physical constraints become relevant. Any large cache by definition is slow, and thus defeats the purpose. Thus, we are stuck with small, fast caches; the question that remains is how to best use them to improve performance.

- Who Handles The TLB Miss?:
- Two answers are possible: the hardware or the OS.
- In the olden days, the hardware had complex instruction sets, called **CISC** for **complex-instruction set computers**, and the people who built the hardware didn't much trust the OS people. Thus, the hardware would handle the TLB miss entirely. To do this, the hardware has to know exactly where the page tables are located in memory via a page table base register, as well as their exact format. On a miss, the hardware would "walk" the page table, find the correct page-table entry and extract the desired translation, update the TLB with the translation, and retry the instruction.
- More modern architectures have what is known as a software-managed TLB. On a TLB miss, the hardware simply raises an exception, which pauses the current instruction stream, raises the privilege level to kernel mode, and jumps to a trap handler. This trap handler is code within the OS that is written with the express purpose of handling TLB misses. When run, the code will lookup the translation in the page table, use special "privileged" instructions to update the TLB, and return from the trap; at this point, the hardware retries the instruction resulting in a TLB hit.
- TLB Contents - What's In There?:
- Let's look at the contents of the hardware TLB in more detail. A typical TLB might have 32, 64, or 128 entries and be what is called fully associative. Basically, this just means that any given translation can be anywhere in the TLB, and that the hardware will search the entire TLB in parallel to find the desired translation. A TLB entry might look like this:


- Note that both the VPN and PFN are present in each entry, as a translation could end up in any of these locations. The hardware searches the entries in parallel to see if there is a match.
- In other bits, there are usually valid bits, protection bits, dirty bits, etc.
- TLB Issue - Context Switches:
- With TLBs, some new issues arise when switching between processes and hence address spaces. Specifically, the TLB contains virtual-to-physical translations that are only valid for the currently running process. These translations are not meaningful for other processes. As a result, when switching from one process to another, the hardware or OS or both must be careful to ensure that the about-to-be-run process does not accidentally use translations from some previously run process.
- One approach is to simply **flush** the TLB on context switches, thus emptying it before running the next process. On a software-based system, this can be accomplished with an explicit and privileged hardware instruction; with a hardware-managed TLB, the flush could be enacted when the page-table base register is changed. In either case, the flush operation simply sets all valid bits to 0, essentially clearing the contents of the TLB. By flushing the TLB on each context switch, we now have a working solution, as a process will never accidentally encounter the wrong translations in the TLB. However, there is a cost: each time a process runs, it must incur TLB misses as it touches its data and code pages. If the OS switches between processes frequently, this cost may be high. To

reduce this overhead, some systems add hardware support to enable sharing of the TLB across context switches. In particular, some hardware systems provide an **address space identifier (ASID)** field in the TLB. You can think of the ASID as a process identifier (PID), but usually it has fewer bits.

- Issue - Replacement Policy:
- As with any cache, and thus also with the TLB, one more issue that we must consider is **cache replacement**. Specifically, when we are installing a new entry in the TLB, we have to replace an old one, and thus the question: which one to replace?
- One common approach is to evict the **least-recently-used** or **LRU entry**. LRU tries to take advantage of locality in the memory-reference stream, assuming it is likely that an entry that has not recently been used is a good candidate for eviction.
- Another typical approach is to use a **random policy**, which evicts a TLB mapping at random. Such a policy is useful due to its simplicity and ability to avoid corner-case behaviors.
- **Paging - Smaller Tables:**
- Simple Solution - Bigger Pages:
- We now tackle the second problem that paging introduces: page tables are too big and thus consume too much memory.
- We could reduce the size of the page table in one simple way: use bigger pages.
- The major problem with this approach, however, is that big pages lead to waste within each page, a problem known as **internal fragmentation**. Applications thus end up allocating pages but only using little bits and pieces of each, and memory quickly fills up with these overly-large pages.
- Hybrid Approach - Paging and Segments:
- Another solution is to combine paging and segmentation in order to reduce the memory overhead of page tables.
- Multi-level Page Tables:
- A different approach doesn't rely on segmentation but attacks the same problem: how to get rid of all those invalid regions in the page table instead of keeping them all in memory? We call this approach a **multi-level page table**, as it turns the linear page table into something like a tree. This approach is so effective that many modern systems employ it.
- The basic idea behind a multi-level page table is simple. First, chop up the page table into page-sized units. Then, if an entire page of page-table entries (PTEs) is invalid, don't allocate that page of the page table at all. To track whether a page of the page table is valid and if valid, where it is in memory, use a new structure, called the **page directory**. The page directory thus either can be used to tell you where a page of the page table is, or that the entire page of the page table contains no valid pages.
- The page directory, in a simple two-level table, contains one entry per page of the page table. It consists of a number of page directory entries (PDE). A PDE minimally has a valid bit and a page frame number , similar to a PTE. However, as hinted at above, the meaning of this valid bit is slightly different: if the PDE is valid, it means that at least one of the pages of the page table that the entry points to is valid. If the PDE is not valid, the rest of the PDE is not defined.
- Multi-level page tables have some obvious advantages over approaches we've seen thus far. First, and perhaps most obviously, the multi-level table only allocates page-table space in proportion to the amount of address space you are using; thus it is generally compact and supports sparse address spaces. Second, if carefully constructed, each portion of the page table fits neatly within a page, making it easier to manage memory; the OS can simply grab the next free page when it needs to allocate or grow a page table.