Dinasour Book Chapter 7 Notes - Main Memory

It is important to note the difference between **memory vs. disk**. Memory is storage which the CPU accesses and interacts with immediatly. Disk is where things like files store long term. Virtual memory simulates the memory slot but is actually stored in disk so it can benefit from the larger size of the disk.

This link has some more information: Memory and disk storage both refer to internal storage space in a computer. Each is used for a different purpose.The term "memory" usually means RAM (Random Access Memory); RAM is hardware that allows the computer to efficiently perform more than one task at a time (i.e., multi-task). The terms "disk space" and "storage" usually refer to hard drive storage. Hard drive storage is typically used for long-term storage of various types of files. Higher capacity hard drives can store larger amounts and sizes of files, such as videos, music, pictures, and documents. "Virtual memory" is hard disk space that has been designated to act like RAM. It assists the computer with multi-tasking when there is not a sufficient amount of RAM for the tasks.

Here is $4 + 4 + 5$ mathematical content:

$$4 + 5 = 9$$

wow!

In the last unit, we learned about concurrency: how computer systems can execute multiple programs (processes) at once on one processor. Now we turn to memory management: how computer systems can manage the execution of multiple programs at once with only one main memory. Because of course, this is necessary to have multiple processes running at once.

In this section, we aim to look at algorithms and schemes that solve this problem.
- CPU can only directly access data from registers or main memory. If something from disk needs to be accessed, it must first be moved to main memory.

7.1 - Hardware of main memory

So what does MM actually look like? Remember, it just a long contigious array of bytes where each byte has an address. It looks like this:

Each process needs to have its own contigious block of memory that it can access. To help manage this (that is, to protect processes from interfering with each other), there are two registers which are called the **base and limit registers** which determine the legal range of

addresses that a process may access (question: is this because processes might move around and are not static?). The above figure shows an example of the base and the limit ranges (remember MM grows downwards so the base is actually higher up in the figure): process can access bytes addressed between (inclusive) base: 300040 and limit: 120900. This scheme is visually depicted as follows:

- Importantly, only OS kernel mode can change the values in these registers: not user programs. So the OS really provides the lubricating jelly here: makes it work for the higher-level processes.
- OS will continiously boot out user processes in concurrency to replace it with another process to execute.

7.1.2 - Address binding

- A "program" resides in disk as an executable. Upon execution, program is brought into memory and becomes process. So processes waiting to be brought into MM for execution are stored in an **input queue**.
- In standard old-school single-tasking, OS would select one process from the queue and load it into memory. CPU that can access code instructions and data from the memory. Then, process terminates and the memory space is now available for the next process to run.

I need to understand address binding a bit better tbh.

7.2 - Swapping

- As we know, a process must be in MM to be executed. However, a process can be **swapped** temporarily out of memory to something called a **backing store** (when it pauses execution) and then brought back into MM for continued execution.
- Swapping makes it possible for the aggregate space of all the running processes (logical space) to exceed that of the physical address space of main memory: thus increasing the degree of multiprogramming. This is all visually depicted like so:
- We will discuss some swapping techniques here.

7.2.1 - Standard swapping

Most intuitive way (first thing you'd think of):

1. Standard swapping involves moving processes (their memory images that is: see here) between MM and a backing store (BS). BS is commonly some fast disk. Must be large enough to accomodate copies of all memory images for all users.

2. System maintains a **ready queue** consisting of all processes whose memory images are on the abcking store or in memory and are ready to run.
3. Whenever CPU scheduler decides to execute a process, it calls the dispatcher. From here, the dispatcher checks to see whether the next process in the queue is in memory. If not, it goes to retrieve it from BS and loads into memory. If there was some previous process in memory, it just does a simple swap.
4. Reloads registers and transfers control to this new process.

Problems: - Standard swapping is a good conceptual introduction but it is not currently used that often because it requires too much swapping time rather than execution time to be an efficient memory management solution. We will talk about some other solutions below but to get you thinking, what happens if we swap only *part* of each process rather than the entire thing? This is kind of the basis of the virtual memory which we will cover next.

7.3 - Contiguous memory allocation

We discussed swapping above to control execution of processes during run time. But we can also study how we can effectively actually **allocate** the process images into main memory. To be more specific, when we swap a process from backing store (disk) to main memory, where should we place the process memory block? How can we do this most efficiently in conjuction with all of the previously placed blocks of memory? This is the idea of contiguous memory allocation which we discuss here. This is a direct mirror of heap allocation techniques discussed in CS 107.

- MM must hold both the OS and all other user processes. So we need to be as efficient as possible.
- We usually place OS in the low memory (the first thing) because of proximity to the interrupt vector which is placed in low-memory.

Memory protection

- Before we even begin to discuss allocation, we should understand the idea of **memory protection**: that is, **how can we prevent a process from accessing memory it does not own**?
- Virtualization: we can do this with the relocation (base) and limit registers.
  - Relocation register: specifies the base physical address of the process.
  - Limit register: specifies the range of logical addresses (those functioning per the CPU and user program).
  - Example: relocation = 100040 and limit = 746000.

  - So each logical address must now fall within the range specified by the limit register. The MMu does the mapping from the logical addresses dynamically by adding the value to the value in the relocation register.

- CPU scheduler selects a process for exeuction and dispatcher loads the correct relocation and limit registers.
- So when the CPU generates new logical addresses during run time, we do a quick check to ensure that the addresses fall between these ranges set by registers.
- This helps make the OS (really?) dynamically sized.

Memory allocation

Ok… now let us get into allocation now that we have an idea for how process memory is protected.

The first idea is that of **fixed-partitions**.

1. Divide memory into several fixed size "partitions".
2. Each partition may only contain one process.
3. When a partition if free, a process is selected from the queue and is loaded into this free partition.
4. When the process terminates, the partition becomes available for the next process.

Pretty easy to see limitations and why it is no longer used commonly: fixed size, can only have as many programs running = number of partitions.

The next scheme is just about identical to heap allocation techniques from 107: **variable-partitioning**. We don't divy up the memory into partitions designated for each process (discretize). Instead, we view the memory as a continious region. At first, the entire free region is one big **hole**. OS takes processes from input queue according to scheduling algorithm. We try to find region in the holes where we can allocate next process. We do this until we no longer have enough free space for the next process. In which case, the OS can decide to stall until there is enough free memory or jump to the next process small enough in the queue (depending on the scheduling algorithm at play). Here is how the textbook describing this allocation scheme:

> In general, as mentioned, the memory blocks available comprise asetofholes of various sizes scattered throughout memory. When a process arrivesand needs memory, the system searches the set for a hole that is large enoughfor this process. If the hole is too large, it is split into two parts. One part isallocated to the arriving process; the other is returned to the set of holes. Whena process terminates, it releases its block of memory, which is then placed backin the set of holes. If the new hole is adjacent to other holes, these adjacent holesare merged to form one larger hole. At this point, the system may need to checkwhether there are processes

waiting for memory and whether this newly freedand recombined memory could satisfy the demands of any of these waitingprocesses.

Sound familiar? This is dynamic storage allocation: **how can we satisfy a request of size n from a list of free holes?**

In it of itself, there are many solutions to this scheme which we all know about from 107. Common strategies include:

- **First-fit**: traverse list of holes. Allocate the process to the **first hole** that is big enough. Search can start from beginning of list of holes or where it left off from previous search.
- **Best-fit**: designate the process to the **smallest hole** that is big enough for the process to fit. For this, we must search the entire list (unless we order the list by size). Produces smallest leftover hole.
- **Worst-fit**: traverse list of holes. Allocate the process to the **largest hole**. Again, must traverse entire list. Produces largest leftover hole. Reasoning is that the leftover hole migfht be more useful for future processes than best-fit small leftover holes.

Fragmentation

Pitfalls of memory allocation that is inherently bound to happen based on the nature of the problem (John thinks that you can never have a perfect solution that avoids memory fragmentation).

Two types (see these notes for more):

- **Internal fragmentation**: an allocated block is larger than what is needed for the process (example: we may only need four bytes but we must start the next sequence of bytes 4 bytes further since we align everything via 8 byte widths). "Caused by the fact that memory is allocated in blocks of a fixed size, whereas the actual memory needed will rarely be that exact size.".
- **External fragmentation**: no single available block is large enough to satisfy the request, but there is, in aggregate, enough free memory available (e.g. what we described above).

Some stats: - **50-percent rule**: one-third of memory may be unusable. Statistical analysis of **first fit** states that given N allocated blocks, another 0.5 N blocks will be lost due to fragmentation. - On average, for every request, only 1/2 of the block will be filled up by the process. So 1/2 of every block is wasted due to internal fragmentation on average.

How to solve these problems? Can never perfectly solve but some ideas we will discuss include segmentation and paging. Basically: - **Coalescing**: If the programs in memory are relocatable, (using execution-time address binding as previously discussed), then the external fragmentation problem can be reduced via compaction (coalescing at runtime!), i.e. moving all processes down to one end of physical memory. This only involves updating the relocation

register for each process, as all internal work is done using logical addresses (not the physical address space). - **Non-contigious blocks of physical memory through logical address mapping**: Another solution as we will see in upcoming sections is to allow processes to use non-contiguous blocks of physical memory (but "virtually" contigious in logical address space?), with a separate relocation register for each block. - Segmentation and paging (the topic of the next two sections acheive this).

Aside: Logical vs. physical addresses

It is vital that you understand address binding and the differences between logical addresses and physical addresses before moving on (as the next sections begin an introduction to virtual memory which involves mapping logical addresses to physical addresses… everything to this point has just been absolute mappings). There was a section above on it, but I think it makes more sense to discuss it here. So before moving on, I will write some notes on it here. These notes are based on this document.

Aside: os-step book states that the "address space" of a process is an abstraction of physical memory that the program sees. It is the e.g. stack, heap, code, etc. It is *not* an image of the physical main memory itself.

The program has no privelege to view main memory physically. The OS does and handles this (and ensures things like protection so that no process can access another processes memory). The main point I want to stress here is that the program's view of memory is isolated to its address space which itself is a virtualization of physical memory (not the entire image of physical memory, but rather that of the process block). The memory management unit is then responsible for translating the addresses from the virtual address space to the physical memory slab. So stress this more, in the above figure of a process's virtual address space, notice that the addresses start at 0 but this isn't the actual address of the code segment in memory. The MMU translates this address. Thus, we consider this a **virtualization** of the memory. Again, what I want to stress is that even basic techniques like base-bound translation are virtual memory because the address space is itself virtual. So we need some form of address translation even in this technique.

- **Address** uniquely identifies a location in the memory (byte-indexed).
- **Logical address**: a *virtual* address and can be viewed by the user.
  - Used like a reference, by the user, to access the physical address.
- The **physical address**, computed by the MMU, is where all the dirty, behind-the-scenes stuff happens.
  - Cannot be accessed by the user.

- Difference between logical and physical address is that **logical address is generated by CPU during a program execution** whereas, the **physical address refers to a location in the memory unit**.

See chart here for this differences:

Pause: you might be wondering, wny do we need this whole virtualized mapping scheme? Why not give user processes direct access to the main memory unit? The answer is security. We don't want applications to be able to overwrite other processes for example. So we let the operating system take this role and it does this by creating a virtual address space for the user and then performs the mapping to the physical address space. To see more about the motivation, see this Stack Overflow thread.

Logical addresses

Let us take a deeper dive on logical addresses.

- It is an address generated by the CPU for a program: e.g. program local variables such as `int x` get a virtual address.
- This address is not a physical memory address. It is used to locate a physical memory address.
- We call the **set of all logical addresses the logical address space**.
- The **Memory-Management Unit** (a piece of hardware) maps the logical addresses to the corresponding physical addresses.
- During compile and load time, the logical addresses are identical to the physical addresses. The real power comes into play during execution time. At run time, the address-binding methods by the MMU generate different logical and physical address.

Physical addresses

Let us now discuss physical addresses in more depth.

- Physical Address identifies a physical location in a memory. It is an address to some byte-index.
- The MMU computes the corresponding physical address for each logical address.
- User cannot access. Kernel mode OS can though.
- The set of all physical addresses corresponding to the logical addresses in a Logical address space is called Physical Address Space.

Diagram:

Key differences

More on the hardware and address binding

We again need to review the beginning parts of this chapter to solidify understanding.

- CPU needs to access main memory to execute a process. Process is stored in main memory and its image includes its code instructions, data, etc.
- CPU can only access direct main memory and not hard drive disk. So all programs must be loaded in main memory to execute.
- Only the OS has access to the physical address space. Logical addresses (for things like `int x` variables) are generated by the CPU for each program.
- User processes can only access memory (via logical addresses) that belongs to it. What designates the memory a process can access (again not physically but by means of virtual logical addresses) is based on two registers which define a range of addresses that the process can access: a base register and a limit register.
- Every memory access made by a user process is checked against these two registers, and if a memory access is attempted outside the valid range, then a fatal error is generated.
- OS has privaleged access to all memory locations as they perform job of swapping. Changing of base and limit registers (dynamically changing size of process?) is only allowed by the OS as well.

Now let us talk about address binding (see this link and this one too) a bit:

> The process of associating program instructions and data to physical memory addresses is called address binding. That is, how can we find the addresses for which the program should live during execution?

- User programs typically refer to memory addresses with symbolic names such as "`i`", "`count`", and "`averageTemperature`". These symbolic names must be mapped or 'bound' to physical memory addresses. This occurs in several stages during the living time of the program:

1. **Compile time**: can generate address for the process to live (load address) during compile time: if it is known at compile time where a program will reside in physical memory, then absolute code can be generated by the compiler, containing actual physical addresses. However if the load address changes at some later time, then the program will have to be recompiled (boo!).
2. **Load time**: say that the location at which a program will be loaded is not known at compile time. In this case, he compiler must generate **relocatable code**. In this case, all of the **addresses are simply in reference to the start of the program** (so that when the load address becomes known, the physical addresses can be calculated by adding the referenced addresses to the load address). So in this case, if that starting address changes, then the program must be reloaded but not recompiled.

3. **Execution time**: sometimes we may actually want to move around a program/process in memory during the course of its execution. In this case, binding must be delayed until execution time. This requires special hardware (i.e. memory management unit), and is the method implemented by most modern OSes.

So we now understand that address binding can occur in one of the three stages above. Addresses bound at compile time or load time have identical logical and physical addresses. Addresses created at execution time, however, have different logical and physical addresses. **This mapping is performed by the MMU at run-time** (how amazing is that! on the fly!). How exactly does the MMU do this? That is what we will discuss next chapter but for now:

- The MMU can take on many forms. One of the simplest is just to modify the base-register scheme described earlier. i.e.:
- The base register is now termed a relocation register, whose value is added to every memory request at the hardware level. We can modify the location of all the addresses by changing the relocation (base) address on the fly and then computing the new addresses by adding the relocation address to it (because everything is in reference the base address (I think?)).
- Note that user programs never see physical addresses. User programs work entirely in logical address space, and any memory references or manipulations are done using purely logical addresses. Only when the address gets sent to the physical memory chips is the physical memory address generated.

Alright… back to the action.

7.4 - Segmentation

We left off with how we can mitigate the effects of fragmentation in contigious memory allocation. We mentioned that we can coalesce blocks of free memory if the addresses are relocatable. However, another solution **is to allow processes to use non-contiguous blocks of physical memory** (but "virtually" contigious in logical address space?), with a separate relocation register for each block. There are two approaches to this and segmentation is the first one we will discuss. This begins the real intro to "virtual memory".

(The abstraction)

The basic idea behind segmentation is to model how a programmer would think. We programmers tend to view memory split up between things like the stack, heap, code, global variables, and libraries. But main memory is just a contigious sequence of bytes. But remember, user's only work with a logical/virtual interpretation of this. So t**he idea of segementation is that the logical address space can be split up into** segments** where each segment corresponds to some semantic meaning (e.g. heap)**.

> Actually, I don't think this is the case. You should view segmentation as an extension/follow-up to the base-and-bounds approach. Instead of giving each process a base and a bound. We are going to divide the processes memories up into segments and then these itself are like base and bounds (so we have an array of base and bounds). The idea is that we can move around these *smaller* blocks more coherently, reduces fragmentation, easier to compact, but the main conveniance is the ability to allow the process memory to be physically discontinious. A note: internal fragmentation *cannot* occur because the segments are allocated based on how the logical memory is divided up. However, external fragmentation can still occur if for example, the blocks don't fit into a space.

So now, the logical address is a two tuple:

$$< \text{ segment-number, offset } >$$

where offset is distance in bytes from base address of the segment.

Segmentation hardware

(The physical realization)

Although the programmer can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one dimensional sequence of bytes. So what is being done on the physical side of things?

- **We must define an implementation to map two-dimensional programmer-defined addresses into one-dimensional physical addresses**. Going from 2d to 1d!
- How does this mapping work?
- Done by the **segment table**.
- Each entry (one entry per segment that is) in the **segment table** has a **segment base** and **segment limit**.
- Segment base: contains the starting physical address where the segment resides in memory (physical memory).
- Segment limit: specifies the length of the segment

The use of a segment table is illustrated in the above figure. - A logical address consists of two parts: a segment number,s,and an offset into that segment,d. - The segment number is used as an index to the segment table (like an array index, to get the correct segment). - The offset d of the logical address must be between 0 and the segment limit. We check for this, and if it is not, we trap (send error) to the operating system (logical addressing attempt beyond end of segment... *segfault*!). When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array

of base–limit register pairs. - So I think… from my understanding, that this means multiple processes can use these sections in a shared fashion and so the process block is not contigious. Is this correct? Like each new process comes in and divides itself by the semantic segments (e.g. puts some of its memory in the heap, has its functions in the stack area, etc.). New process does the exact same thing. **The segments are shared between processes**.

Pictoral example:

See how we now have physical "segments" in the physical memory rather than process blocks?

Some auxillary notes from the course lecture notes:

- Each segment also has a protection bit for each segment that denotes whether it is read-write versus read-only.
- Segmentation table (map as its called in the course notes) is itself stored in the MMU.
- Table is modified during context switches
  - See Ed question here. Basically, not all segments are shared between processes. Some might have function segments unique to a process so it takes up its own segment. In these cases, some of the segments are swapped out of the table and physical memory during context switching.

- In practice, we the two-tuple referencing scheme can be stored within bitwise semantics: Top bits of address select segment, low bits indiciate the offset.

Pros and cons of segmentation

Pros: Flexibility. - Permits the physical address space of a process to be non-contiguous (main motivation) - Each segment is managed independently. - e.g. stack doesn't interfere with text segment (unless you get a pesky segfault). - Grow/shrink independently - Swap to disk independently when needed - Can share segments between processes - e.g. helpful if you want to do something like share code - Can compact/coalesce memory to reduce fragmentation

Cons: - Limited to fixed number of segments - Fragmentation - But how? - Interestingly, fragmentation can occur on the backing store during swaps (because colaselcing isn't possible on backing store since CPU cannot access it) - Maybe internal fragmentation (like some processes won't have enough code to fill up the text segment) - Virtual address space is rigidly divided

7.5 - Paging

We now introduce the most used method for memory management in today's systems: paging. It is so good because it is virtual and it solves the problems of the other virtual memory manangement solution (segmentation).

- We saw that segmentation **permits the physical address space of a process to be non-contiguous**. Paging also offers this pro.
- The difference, is that **paging avoids external fragmentation** and also avoids the need to compact/coalesce.
- Another pro is that it solves the considerable problem of fitting memory chunks of varying sizes onto the backing store. Most other schemes cause fragmentation on the backing store itself (when needed to swap, it is essentially the same problem (allocation) but in a different direction lol).
- So… what is paging, how do we accomplish and implement it? Well, we are going to need some support from the hardware and it is implemented in the OS.

7.5.1 - Basic methods

- Paging is a memory management scheme that **allows process's physical memory to be discontinuous**, and which **eliminates problems with fragmentation** by allocating memory in equal sized blocks known as pages.
- Idea is to break up physical memory into fixed-size blocks called **frames** and break logical memory into blocks of the same size called **pages**.
- So the idea is that when a process is to be executed, during the loading process (I think?) its pages fill in the available memory frames from their source (backing store or physical file).
- Not only that, the backing store itself is divided into the same fixed-sized blocks that are the same size of the frames.
- It is easy to see that the logical address space (pages) is now completely separate from the physical address space. Because of this, we can simulate to the user that there is more memory than there might be physically available.

In summary from above: - Paging involves breaking physical memory into equal fixed-sized blocks called **frames**. - Then, the logical address space (virtual memory displayed to the programmer) is itself broken up into the same fixed-sized blocks called **pages**. - When process executed, load pages into available frames. - Advantages: allows process's physical memory to be discontinuous, more virtual memory than physical memory, avoids fragmentation

Hardware support: - When CPU generates a logical address, it is divided into two parts: - **Page number (p)** - **Page offset (d)** - We use the page number to index into a **page table**. - The **page table** is table containing the **base address of each page in physical memory** - This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

Page size

Let us talk about the size of the pages. Dependent on hardware. Typically by powers of 2 because then translation from logical –> physical is easy because we can user high-order bits.

- Say that size of logical address space is 2^m bytes and we have page size as 2^n bytes. This means that the *offset* itself will only take up n bits. So we can use the remaining $m - n$ bits to store the page number and use those n bits to store the offset into that page.

See textbook for example of how to index into a page table to get the physical address. The general idea is conveyed through the following visual:

back to section

- Importantly, there is **no external fragmentation**: any free frame can be allocated to some process that may need it.
- Though **internal fragmentation** can still occur: but this only ahappens on the last page possible because the memory will be divided into the fixed-size page blocks. For example: > Notice that frames are allocated as units. If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full. For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of 2,048 − 1,086 = 962 bytes. In the worst case, a process would need n pages plus 1 byte. It would be allocated n + 1 frames, resulting in internal fragmentation of almost an entire frame.
- Because of this, on avg, expect 1/2 of page of internal fragmentation per process.
  - So we should maybe try to make pages as small as possible then? Maybe but maybe not: overhead of keeping that many pages increases. For example, disk i/o (backing store swapping) is faster with larger pages because less pages have to be transferred over.
- In today OS, we generally use 4 KB page sizes.
- I think the general idea is that once a process requests memory (i.e. it begins its execution and the code is loaded into memory), it builds a page table for that process, necessary free frames are allocated and hold the processes memory (the frames that are free are stored in a free-frame list) –> then these frames are inserted into the initialized page table corresponding to that process.
- These notes say
  - "Processes are blocked from accessing anyone else's memory because all of their memory requests are mapped through their page table. There is no way for them to generate an address that maps into any other process's memory space."
  - But I am still confused on how this provides protection? Why can't a program/cpu generate a bit address whose page number bits go to a page beyond what is stored in the page table?

- The **operating system must keep track of each individual process's page table**, **updating it whenever the process's pages get moved in and out of memory**, and applying the correct page table when processing system calls for a particular process (?). This all increases the overhead involved when swapping processes in and out of the CPU. (The currently active page table must be updated to reflect the process that is currently running.)

Frame table (optional)

- Ok… so we know that each process has a page table. And we know that the OS is responsible for all of this handling. But how does it actually know what frames are available at what time and when to give some of the frames to pages of processes? It must manage all of this and it does so through a data structure called the **frame table**.
- FT has entries where each one corresponds to a physical frame in memory and indicates:
  - Whether the frame is free or not
  - If not free, which page of which process has the spot.

7.5.2 - Hardware support

> Page lookups must be done for every memory reference, and whenever a process gets swapped in or out of the CPU, its page table must be swapped in and out too (in the MMU?), along with the instruction registers, etc. It is therefore appropriate to provide hardware support for this operation, in order to make it as fast as possible and to make process switches as fast as possible also.

Wait: **note that the page map itself is stored in continuous physical memory**. The PTBR (see below) determines which process page table we would point to.

- Some OS have one page table per process and might for example, store it in the PCB. Others might share page tables across processes to reduce the overhead.
- How can we implement page table in hardware though?
- One idea is to dedicate registers for page tables. But this of course is only possible when we have a small number of pages as registers are limited.
- But of course, today's page tables are massive ($>$ 1 million entries). So the register approach is infeasible.
- So another idea is: **keep the page table itself in main memory** and **have a special register called the page-table base register (PTBR), point (i.e. store a pointer) to the page table in memory**. So whenever we need to change the page table, we just have to change the pointer in this one singular register! Context switching time is reduced.

- But… timing is still inefficient: because **every memory access now requires two memory accesses** - One to fetch the frame number from memory (i.e. go to page table) and then another one to access the desired memory location given by the table. So time efficiency is reduced by power of 2 (not going to work in high circumstances).
- So we got a problem and how we gonna fix it?
- Present day solution is to use a special, small, fast-lookup hardware cache called a **translation look-aside buffer (TLB)**.

Translation look-aside buffer

- Basically, we create a small hardware cache (buffer array/dictionary) of recent translations.
- Each cache entry stores two things:
  - The page number portion from a virtual address (i.e. the page that the CPU wants) is stored as a key and the corresponding physical page/frame number from physical memory is stored as the value.
- So on each memory reference, compare the page number from the virtual address with the virtual page numbers in every TLB entry (which is neatly done in parallel so way more efficient than just a straight memory reference). If the page number is found, its frame number is immediately available and is used to access memory. No memory access needed!
- But… if the page number is not in the TLB (known as a **TLB miss**), a memory reference to the page table must be made :(.
  - At this point, we swap one entry from the TLB with these new translation so that it will be found quickly on the next reference. (optional) Different replacement policies can be used at this step (i.e. least recently used is the one that is swapped out). Also we can "wire down" certain TLB entries that cannot be swapped out such as those belonging to the OS.
  - You might be asking how a memory reference to a page table is actually made if we don't just swap the page table in and out for each process like we had before this sub section. Basically, each page table for each process is stored in the PCB which itself is stored in the OS's memory. The PTBR register holds a pointer to this place. So on a TLB miss, we need to use the pointer stored in the PTBR and access the page table their (1 memory reference) and then actually get the corresponding page/frame wanting to be addressed (another 1 memory reference).
- So on a miss, the process is still the same amount of time (two memory references: PTBR ptr –> page table –> frame). But on a hit, only one memory reference needed!
- But luckily, computers usually have a $> 95\%$ hit ratio. This makes sense. In programming, we define some things like an array that we continue to access throughout the course of the program. So it is best if we keep this memory translation in a cache. The TLB supplies such functionality.

- If you are paying close attention, you might notice on problem with protection: if the TLB remains the same on context switching, then doesn't this mean we give the opportunity to a program to maliciously get some random page from the cache buffer and use it as if it was their own? To protect against this, some TLBs store address-space identifiers, **ASIDs**, to keep track of which process "owns" a particular entry in the TLB. This enables entries from multiple processes to be stored simultaneously in the TLB without granting one process access to some other process's memory location. Without this feature the TLB has to be flushed clean with every process switch if protection is wanted (which it is).
- **Side note**: we aren't actually learning about ASIDs. Instead, for every context switch, we invalidate the entire TLB by setting each entry of the TLB to be invalid (cannot be accessed).
  - Question: So for every context switch, we invalidate the entire TLB? How do we share pages between processes then from the TLB? Also, isn't this inefficient?
-

Visual:

Some various notes: - Lots of math involved with the bit calculations. To determine how mnay bits we need to address a page table of size 2^n bytes, we need n bits.

7.5.3 - Protection

- We can also provide extra protection via the page table. We can allocate a few more bits in the page table for each entry.
- We can use these **protection bits** to specify whether the page can e.g. be read-only, write-only, execute-only, or any combination.
  - Cause OS to issue trap is CPU trying to reference a page to which it shouldn't (i.e. trying to write to a read-only page).

- Importantly, we can also provide a **valid–invalid bit**.
- When this bit is set to valid,the associated page is in th eprocess's logical address space and is thus a legal (or valid) page. When the bit is set to invalid, the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid–invalid bit. The operating system sets this bit for each page to allow or disallow access to the page.

Example (figure 7.15):

- Notice that process doesn't have access to pages 6, 7. So if the CPU somehow generates some address corresponding to that page, OS will issue a trap due to the valid-invalid bit at those entries for this process's page table.

Differences between paging and segmentation

Main difference is that paging is fixed sized blocks and segmentation is variable sized. Paging has only internal fragmentation whereas segmentation has only external fragmentation.

Note: make pros and cons list.

Segmentation pros: - No internal fragmentaion

Paging pros: - No external fragmentation - Virtual memory can be larger than physical memory