Chapter 8 – The address translation:

# **8.1 – Address Translation Concept**

**What is the address translation?**

As mentioned earlier. The operating system provide the illusion of infinite memory. A process sees only the virtual memory. It doesn’t need to worry about how much memory is on the computer, and/or how much memory other processes uses.

Therefore, we need an way to translate the virtual addresses into physical addresses. The addresses must exist in real life.

We can think of the address translation as an black box where the virtual addresses comes in and are then translated into physical addresses.

**What do we want out of address translation?**

* Memory protection
  + A program should only access the memory region that belongs to it. Nothing else.
* Memory sharing
  + We want multiple processes to be able to share some memory between them.
* Flexible memory placement
  + We want to allow the operating system to place a process’ memory anywhere in physical memory. So there is no restrictions.
* Sparse addresses
  + Addresses can grow.
* Runtime lookup efficiency.
  + Everytime we fetch an instruction or data it must be efficient and not take too long time.
* Compact translation tables
  + The translation tables should be small. Not take too much storage
* Portability
  + It should work on most type of architectures.

# 8.2 – Towards Flexible Address Translation

**How does the concept with address translation with base and bound work?**

With the use of base and bound, the processor will send an virtual address. The virtual address is added to the base, and will then be generated to a physical address than can access physical memory. At the same time, the virtual address will be checked against the bound. If it greater than the bound we will raise an exception because it is trying to access memory outside of its region.

Each process has a base and bound register which is essential for example for memory protection, process isolation.

**Why doesn’t every computer use base and bound?**

Even though it is very simple and fast, it lacks many of the features we want the address translation to have. Processes can’t share memory and more

**With virtually addressed base and bounds, what is saved/restored on a process context switch?**

We need to save the base and bound of the process. In other words, we need two save two registers (base and bound).

The base is added to the virtual address from the processor, and if we have a new process we need to update both the base and the bound. This is because of process isolation.

**What are the cons of base and bound?**

We can’t specify the access permissions of the memory (read, write etc.). So therefore, we can’t prevent a program from overwriting its own code.

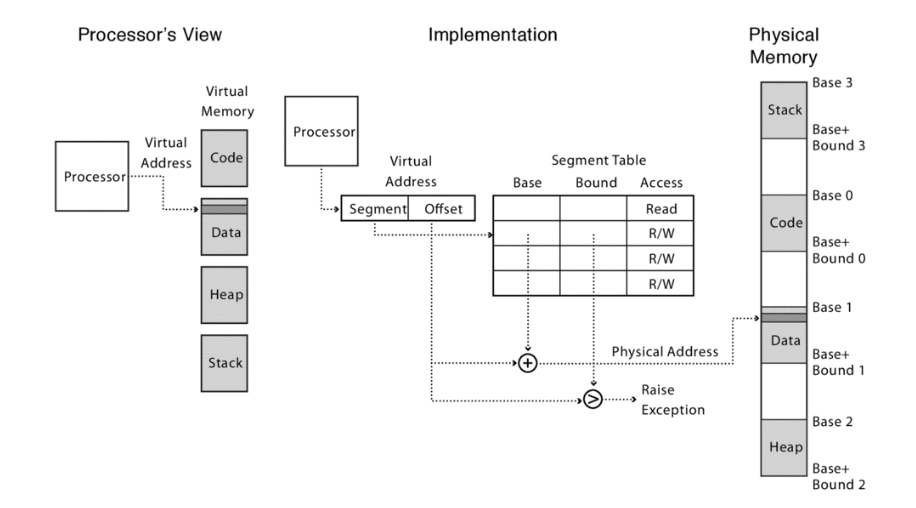
It is not possible to share code with other processes. This is not good as sharing code with other processes is one of the goals with address translation.

We can’t grow the stack or the heap as needed.

**What is segmentation?**

Segmentation is the extended base and bound-concept. We still use the base and bound-idea with segmentation.

With segmentation every process has a segment table. The virtual address has two components; segment and offset. The segment number indicates where in the segment table it points to, and the offset.



The offset is added to the base, and checked against the bound. It raises an exception if the offset is greater than the bound. In addition, the operating system can give each segment permissions restrictions (read, write, read and write etc.).

If checking the offset against the bound doesn’t raise an exception, we get our physical address.

**Why is segmentation preferred over for example base and bound?**

Because with segmentation, we accomplish more of the goals that we wanted with address translation. For example, processes can share memory if they have the same start, length and access permission. Segments can also be stored anywhere in the physical memory.

With base and bound, we did not have the access permission for code. With segmentation, each segment has access permission.

At the same time, segments can be located anywhere in memory.

I think that with base and bound we store the memory of the process as a whole. What I mean by that is that it saves the heap, stack, text and data. But with segmententaton, we divide the memory of the process into segments. Each segment may be store anywhere in physical memory.

**With segmentation, what is saved/restored on a process context switch?**

We need to store the entire segment table. As mentioned, each process has a segment table containing different segments.

**What about UNIX Fork, and copy on write? How does this work with segmentation?**

First of all, UNIX fork creates a completely copy of a process (child). In theory, that would occupy twice as much memory as only having one.

Thinking about the optimization of segmentation, how does this work? We would copy the entire segment table of the parent into the child process. So they point to the same physical memory, which means they **share** memory.

We will then mark the parent and the child segments as read-only. We do not want to write to those segments.

Then we will start the child process and return to the parent. The child will return zero, and the parent will return the child pid.

Then, if either the parent or child write to one of the segments, we will trap into the kernel with a system call. Make a copy of the segment table and resume.

**What is the Zero-on-Reference concept?**

The case when we don’t know how much physical memory is needed for the heap or the stack in advance. It can grow. We only use how much is currently in use.

However, if we try to use more (beyond the end of the stack) we get an **segmentation fault** into OS kernel. The kernel will then allocate some more memory.

When this happens, the kernel **zeroes** out the memory because it doesn’t want to leak any information that was there earlier.

Then, we will modify the segment table because we now have more memory, and the segment table needs to be up to date. Then we resume the process.

**How does segmentation prevent information leaking?**

When the operating system decide to reuse memory or disk space that previously had been used it zero out the content of the memory or the disk. ^^

**Physical memory will over time be divided into several regions of segments. How can we fit in new memory between the regions?**

By external fragmentation. That means that the operating system can chance the physical addresses and make room for new memory because the virtual memory is not affected. And it is the virtual addresses the programs sees. So they will not notice any difference.

**What are the pros of segmentation?**

We can share code/data segments between processes by sharing the same segment number in the virtual address.

In addition, it is possible to protect code segment from being overwritten since we have access permission of each segment.

It is possible to transparently grow/stack as needed.

**What are the cons of segmentation?**

The memory management is quite complex. We need to find chunk of a particular size to put data into memory.

We need to rearrange memory from time to time because over time there will be small gaps/holes of free memory space which makes it hard for other segments to fit in because of the size.

**What is paged translation?**

With segmentation we say that each contiguous segments were not fixed sizes. They all depend on the process. This will led to these holes of unusable memory spaces.

With paged translation, we manage memory in fixed size/units / pages.

Each segment (heap, stack etc.) will be divided. It doesn’t need to be contiguous.

Instead of segments, we have an page table. The page table contains pointers to frames and have the access restriction. Memory is allocated in fixed-size chunks called page frames which are the frames the page table contains pointer to.

The virtual address consists of the page number and the offset. The page number indicates in which page in the page table we are going.

That particular page number in the page table refers to the physical page frame in memory.

!Page frame is in physical memory

!Page number is in virtual memory

So process’ segments are divided into pages in virtual memory. The virtual address consists of two components; the page number and the offset. The page number points to the page in the page table which then points to the specific frame in physical memory.

**What is saved/restored on a process context switch?**

We need to save a pointer to the page table because the page table is stored in physical memory. Therefore we do not need to copy the entire page table since it is always there.

**What if the page size is too small?**

Then we need to have a much bigger page table. We need to divide the same amount of memory into smaller chunks.

**What if the page size is too large?**

**???????**

**Can we share memory between processes with paged translation?**

Yes, by setting the pointer to the page frames in physical memory. So an entry in the page table must point to the same page frame in physical memory.

Therefore, we need a *core map* to store which processes are pointing to which page frames. So it is an data structure to manage the frames, and see who uses those frames in physical memory.

**How does it work with copy-on-write and paging?**

*Copy-on-write:* Processes may share data. If someone tries to modify that data, we make a copy of that data.

With UNIX fork we create a complete copy of a process. How does it work with copy-on-write in this instance? First of all, they share the same memory which means that their page numbers in their page table all points to the same page frames in physical memory.

With copy-on-write we copy the page table of the parent into the child. Mark all pages (both the new and the old page table) as read-only. Trap into kernel on write with a system call. Copy the page, and mark both of them as writeable. Now, both the parent and the child can write to their memory. Then we resume execution.

**What is the difference between paging and segmentation?**

As mentioned, free-space allocation was not that easy with segmentation. With paging every page frame has a fixed size. The operating system can represent physical memory as a bitmap, and then it is only a matter of finding a free frame. Each bit represents a physical page frame that is either in use or not.

So finding a free frame is all about finding a free bit.

**Can processes share data with paging translation?**

Yes. By setting page table entry for each process sharing a page to point to the same physical page frame.

**What is fill on demand?**

*Can I start running a program before its code is in physical memory?*

How do we do that? We start by setting all page table entries to invalid. Then, when a page is referenced for the first time we kernel trap and the kernel brings in the page from disk. Resume execution, and the remaining pages can be transferred in the background while the program is running.

**What is sparse address space?**

**What is paged segmentation?**

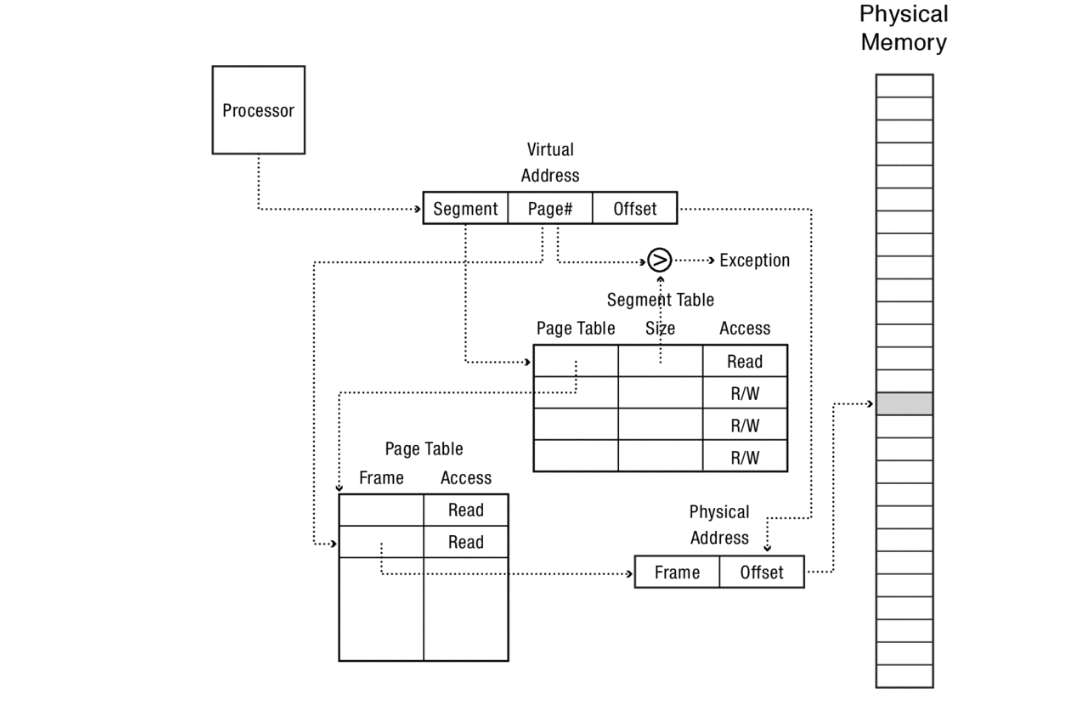
Paged segmentation is a method combining both paged address translation and segmentation.

We have an segment table entry containing pointer to page table, page table length (number of pages in segment) and access permissions.

The page table entry contains a pointer to the page frame in physical memory and access permissions for that frame.

How do we know which page table index is correct? By looking at the virtual address which consists of segment, page number and offset.

The segment number indexes to the segment table which then indexes to the page number in the page table. The page number in the virtual address is checked against the size in the segment table. If is greater than the size we raise an exception.



**With paged segmentation. What must be saved/restored across a process context switch?**

We need to store the segment table per process. The page table is saved in physical memory so we only need to store the pointers to the page tables which lies in the segment table in virtual memory.

**8.4.2 Needs to be read further. I didn’t understand much of it**

## Portability??

# 8.3 Towards efficient address translation

**How do we improve the efficienty towards address translation? How do we make it go quicker?**

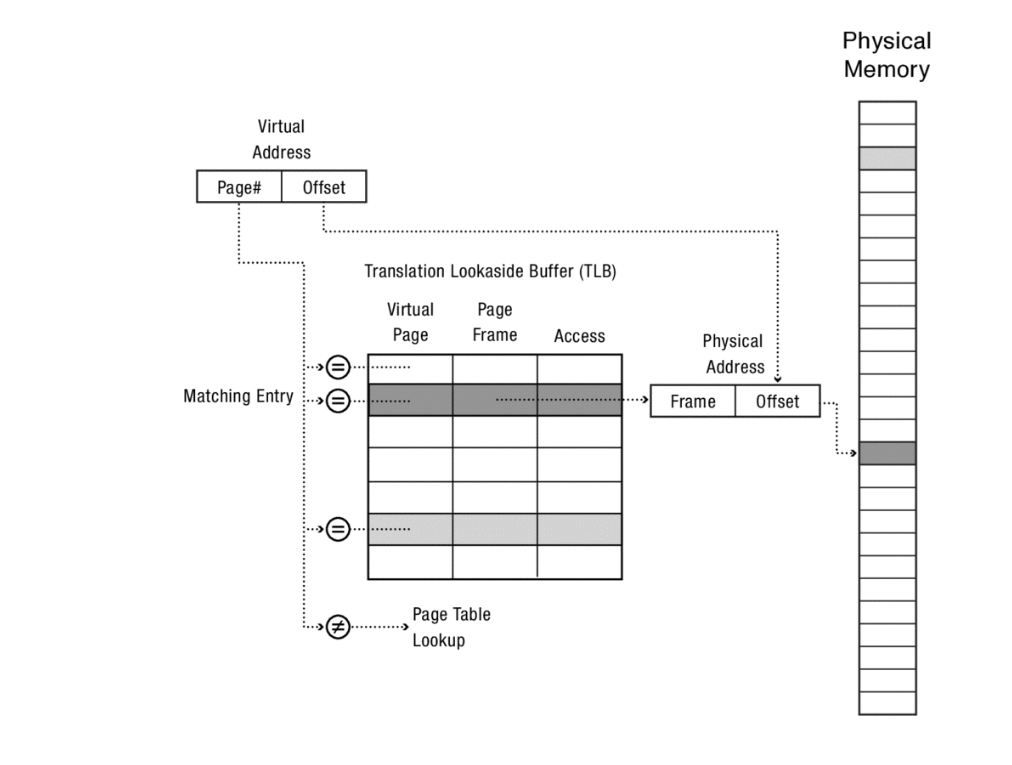
We use what is called cache. A cache contains a copy of some data that can be accessed more quickly than the original.

**Why do we want efficiency towards address translation?**

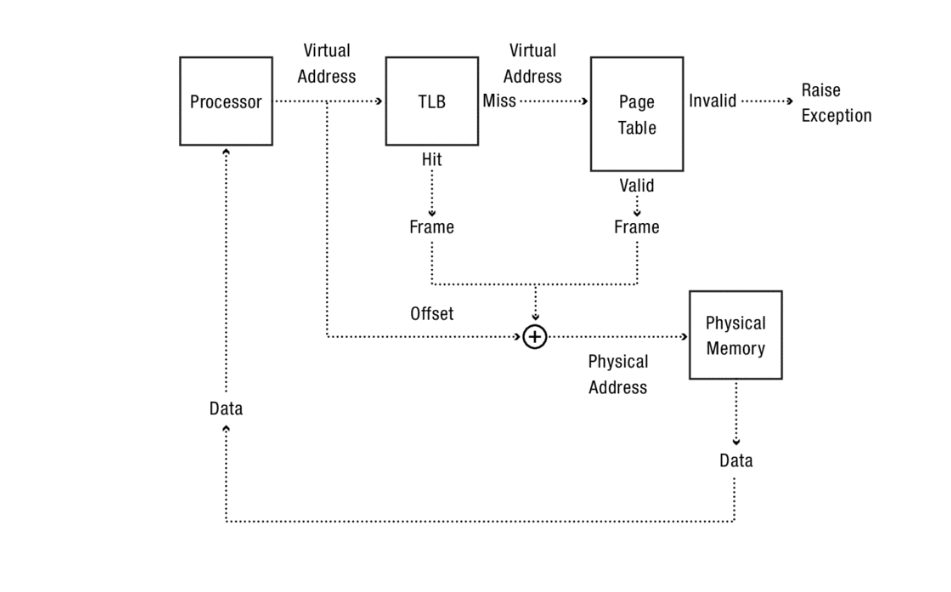
Because, if we need to convert many of the same virtual addresses to physical addresses to a process. That takes some time, and there could be more efficient way of doing this. We use what is called a cache. A cache contains the result of recently used virtual pages -to physical frames translations. It will basically hold a table of recently used translations.

**What is the translation lookaside buffer (TLB)?**

The TLB is a small hardware table containing the results of recent address translations. Each entry in the TLB maps a virtual page to a physical page.



As the picture shows. Each virtual page number is checked against all of the entries in the TLB at the same time. If there is match, the matching table contains the page frame and the access which then will be translated to physical address.

If there is a miss, the hardware multi-level page table lookup is invoked. 

If we find the translation in the table (cache) we will use that translation. If it miss we must walk the multi-level page table. In other words it takes longer time.

**What is the cost of translation?**

Cost of TLB lookup = Prob(TLB miss) \* cost of page table lookup.

**When do TLBs work / not work?**

What happens on a context switch? Do we reuse the TLB, do we discard the TLB? The TLB is populated per process, so it contains addresses of a certain process. Therefore, it not a good solution to reuse the TLB.

The solution is to have a tagged TLB. Each TLB has process ID. TLB hit only if the process id matches current process.

## 8.3.2?