Lecture notes Virtual Memory & Super pages

Why virtual memory:

In modern computers, we have multiple processes running concurrently, and we only have one main memory. Hence, we must make sure that processes access different parts of the main memory, otherwise they would step on each other's feet while executing.

The challenging part is that when processes are programmed, they don't know how many other processes will run while they are running. Perhaps their program will be the only one running, and hence it could use the whole main memory address space. On the contrary, if there is another process running, it must know where that other process saves data in main memory so that it uses a different part. While possible, that creates a lot of work for the programmers.

Virtual memory helps because it manages to solve this issue, while making the developer's life very easy, as they won't have to worry about any issues related to main memory sharing. The OS will do all the work.

Mechanism

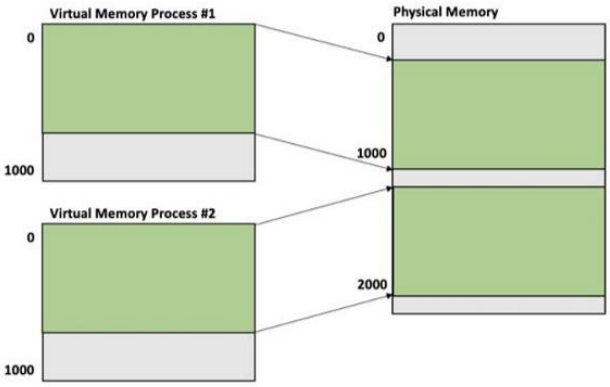
The basic idea behind virtual memory is that every memory address inside a process is considered as a "virtual" address. This address is not necessarily the actual address in main memory. The OS will always translate the "virtual" addresses of a process to an actual "physical" address in main memory. This translation is done per process. Hence, intuitively we can see that thanks to this mechanism we can make it so that if there is a "clash" between two processes, we can simply translate the virtual addresses of one process so that it maps to another area of main memory that is free.

Advantages

Virtual memory helps us to provide isolation and protection across processes:

- **Isolation**: each process has its own address space in main memory.
- **Protection**: processes cannot directly access main memory using physical addresses.

As we can see in the picture below, the virtual memory of process #1 is mapped to a different area in main memory compared to process #2.



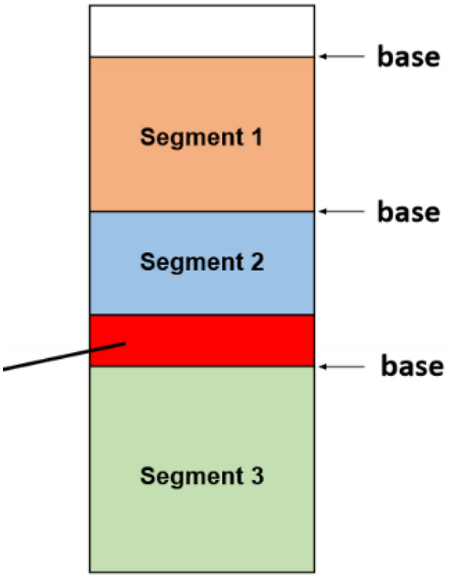
Additionally, virtual memory can enlarge the physical memory capacity by using an external storage to "swap" data in and out, transparently to the processes running.

Implementations

There are two main ways to implement virtual memory:

1-Segmentation

In this implementation, we split the physical memory into varied sized segments, and each process can only access memory within its segments. A segment is defined by its “base”, which is its starting address in physical memory, and its “size”.



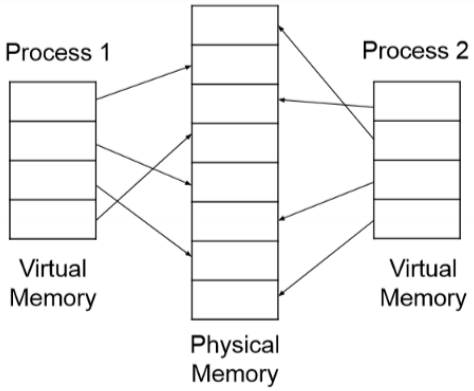
An advantage of this approach is that there should be little internal fragmentation. All the memory of a segment should be used by the owner process.

The first downside of this approach is that the memory withing a segment needs to be continuous. Segments can’t be split into smaller pieces or interrupted by another segment.

Another downside is that this approach suffers from external fragmentation. After some time, when a lot of processes started and finished, we should see small areas between segments that are free but too small to be used for a new segment (as indicated in red in the figure). This leads to a lot of wasted memory.

2-Paging

In this implementation, physical memory is split into chunks of fixed size, called Physical Page Frames (FFP) (middle column in the figure). Then, the OS will map the same size chunks of virtual memory to a PPF that is free.



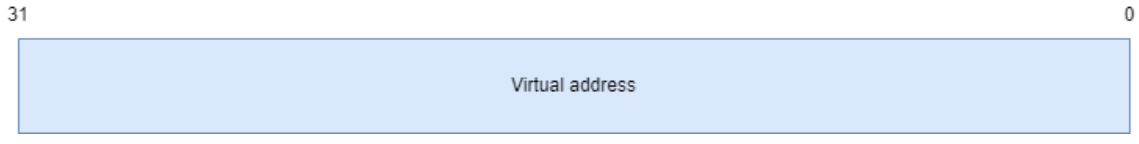
The advantage of this approach is that there is no external fragmentation as a new process can always have some of its virtual addresses be mapped to the free PPF.

Also, this approach eliminates the need for a contiguous area of physical memory, since the address space of a process is broken down into multiple PPF.

The downside is that there can be internal fragmentation inside the PPF, as processes could sometimes need less than the capacity of a PPF.

Paging explained

For the reasons explained above, modern OS now use paging. Here we will explain this method in more detail. Here is a virtual address of 32 bits that a process uses:



If the OS decides to have 4KB PPF, then we need 12 bits to know the offset within that page. The rest then becomes the virtual page number (VPN):

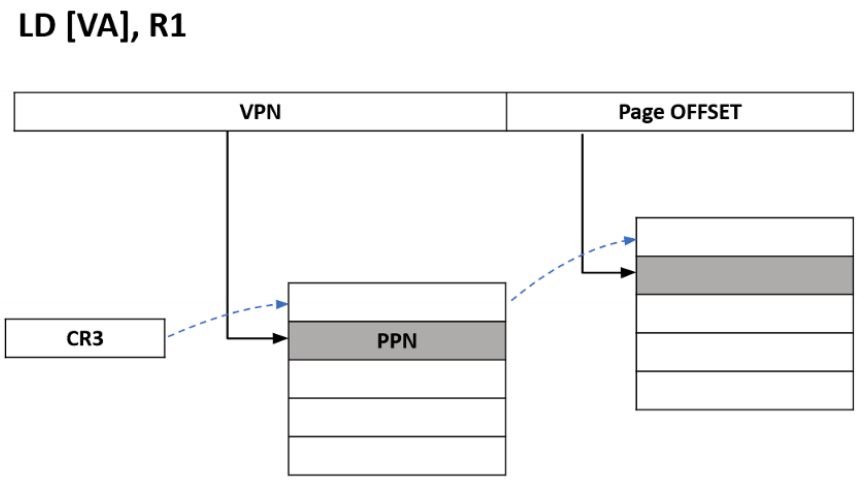
In this case, the virtual page number is 32-12 = 20 bits long. Now, the OS will translate that VPN into a physical page number (PPN). Imagine we are using a 32 bit system, then the OS would have to translate those 20 bit VPN to a 20 bit PPN for each process.

Page table array

In order to do this translation, the OS keeps a datastructure called a **page table**. The first way to implement this data structure would be to use an array, where the index in that array is the VPN, and the value in the array is the PPN. We can also store a valid bit to indicate whether that PPF was swapped to disk. The OS can then modify this array so that the process uses different parts of the main memory.

VPN (index)	PPN	Valid?
0	1501	1
1	12	1
2	434	1
3	51	1
4	52	1
...		
1048574	63463	0
1048575	376	1

The translation process would start at the CR3 register. That register contains the pointer to the start of the page table for the running process. Then, the VPN is used to index into that table, and finally the physical address is constructed by replacing the VPN with the PPN in the table.



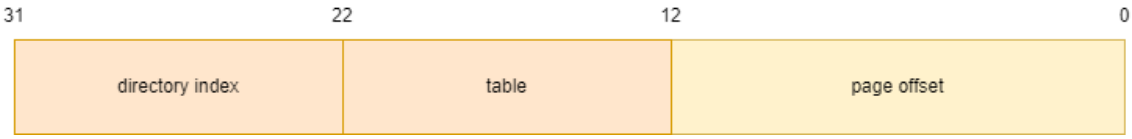
The advantage of this approach is that we only need one memory access to get the mapping.

The downside of this approach is its memory footprint. The size of the page tables is fixed, even if a process uses little memory. In our example, as we have 20 bits for the VPN, we need 2^{20} = 1048576 rows in the table. This would equate to around 4 MB per table, per process. Since we

can have a lot of processes running, the amount of memory taken by the page tables would be too big.

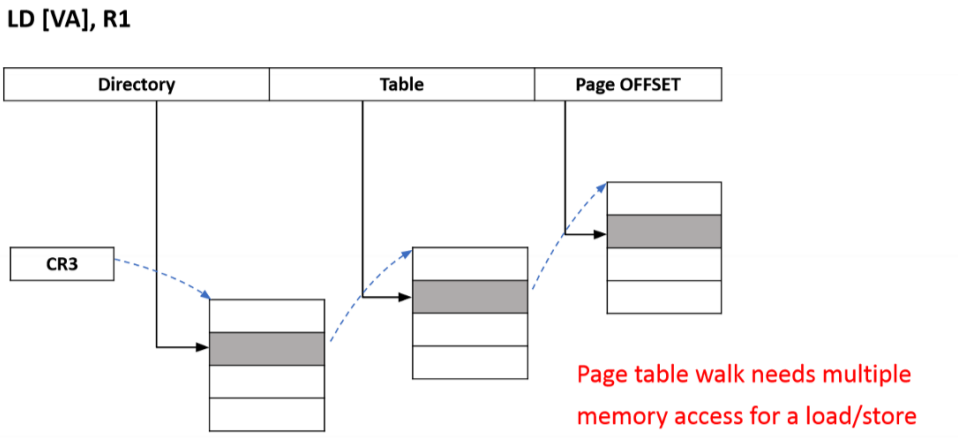
Multi-level page table

We can improve on the previous method, by dividing the single index into multiple ones, which will create a hierarchy of tables:



The top 10 bits would be an index in an array (that we call directory), which will point to another array that we can index with the remaining 10 bits of the VPN. The second array will contain the PPN for the translation (along with the valid bit, etc).

The translation process is similar to the case with a single array:



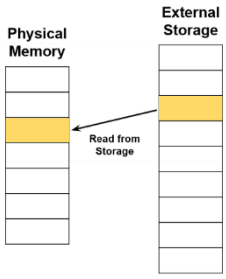
One downside with this approach is that we must perform multiple memory lookups (depending on the depth of the page table).

The major advantage of this approach is that if the process doesn't use a certain range of the VPN values, then in the directory of the page table, instead of a pointer to a new array, we can simply have NULL. This means we don't have a secondary array for that range. In essence, the multi-level page table will only contain mappings for VPN that are actually used by the process. This can save a lot on the memory footprint. Because of this advantage, this is what modern OSs use.

Page Fault

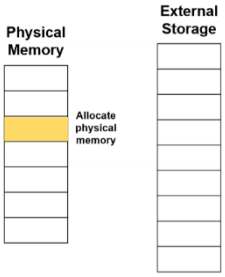
A page fault can arise when there is no PPN for a given VNP in the page table of a process. This event is triggered when the MMU walks the page table to find the translation. In that case, the control is taken from user space and given to the kernel, so it can service that page fault. A page fault can be distinguished in two kinds:

1-Major Page Fault



This page fault is triggered when the kernel previously decided to swap that PPF to disk, to gain space in main memory. Hence, if the process now requests it, we need to execute a slow IO operation to get that data from disk back to main memory. This can take a lot of time.

2-Minor Page Fault



The other case is when we simply need to operate in main memory, and hence such page faults should be much faster to resolve. The first case would be that the data is in DRAM but not yet mapped, so for instance when using shared memory, or if a fork was created then copy-on-write would be enabled. The second case is when the mapping is created but the memory is not yet allocated, for instance when using a deferred allocation of mmap().

Translation lookaside buffer (TLB)

In order to speed up the translation of VPN to PPN, the hardware has a cache dedicated to this translation, called the “translation lookaside buffer”. One thing to note is that each core on a CPU has its own TLB, and hence the OS must make sure that coherence is maintained.

The OS has to mechanisms to maintain his coherence:

- **TLB Flush**: this clears all entries in the TLB of the core.
- **TLB Shootdown**: clear all TLB entries of other CPU cores who have the same mapping. (this is expensive as it uses IPI operations).

Page sizes

It is important to note that the OS must decide on the size of the pages. There is a trade-off. Smaller pages means that you have less internal fragmentation. However, smaller pages means you have more of them to cover main memory, and hence you have more entries in the page tables, and hence with a fixed size TLB you will have more cache misses. On the contrary, bigger pages means more internal fragmentation, but fewer pages and hence smaller page tables and more TLB hits. Choosing an appropriate value for base pages then becomes important for the performance of the OS.

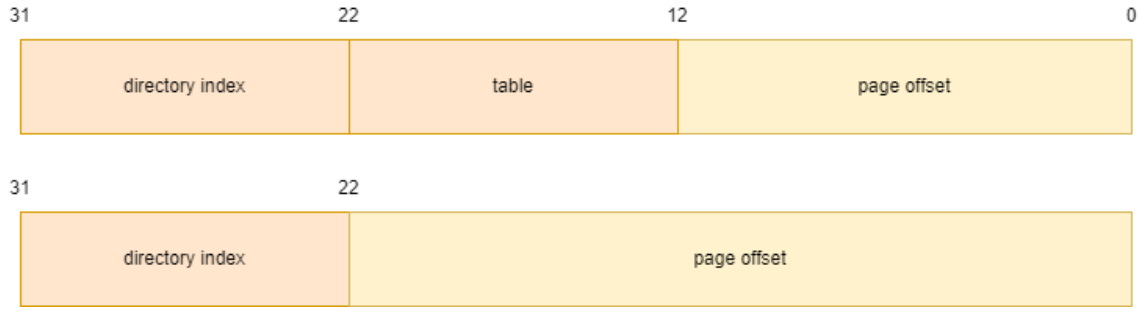
TLB with 512 entries:

- cover 2MB physical memory with 4KB page
- cover 1GB physical memory with 2MB page

Super Pages

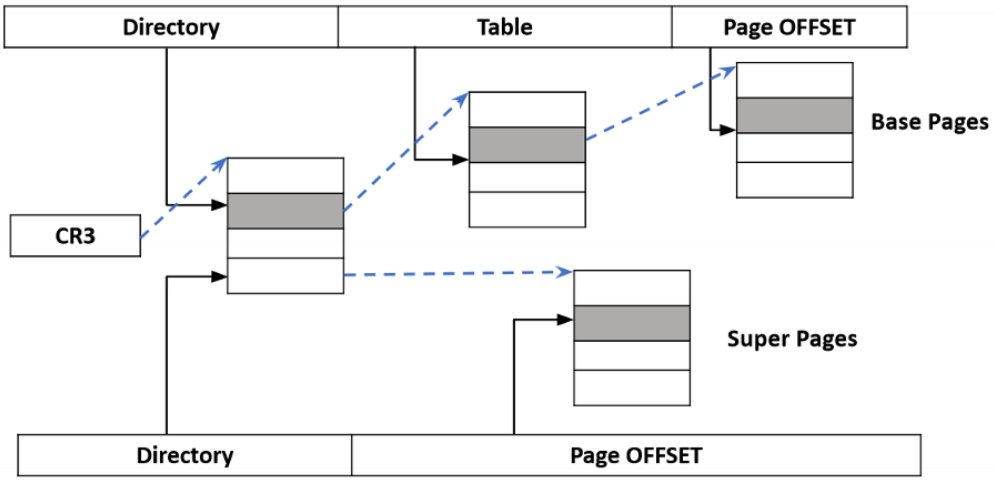
Taking into account the fact that page sizes are important, we can improve on the multi-level page table to try to accommodate pages of multiple sizes. The pages of the default smallest size are “base” pages, and pages of larger size are “super” pages.

Going back to the structure of the multi-level page table, the solution is to use the directory index value as the super page index:



In this case, certain addresses would be used for base pages, and others would be used for super pages. Here, super pages are 2^{22} = 4MB big (from their offset).

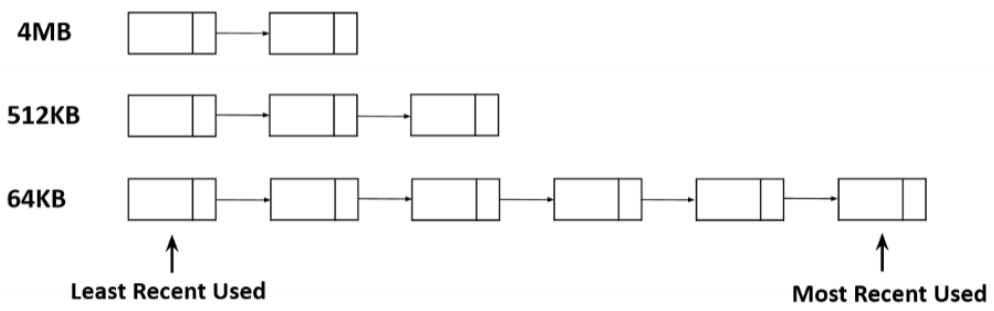
When you want to interpret a virtual address, you first look in the root array using the directory index, and look at the value. It can be referencing a secondary array (in which case the address is in a base page) or it could directly reference a physical super page.



This the advantage with this technique is that the OS can support both small and large pages, depending on the access pattern of the process. Do note that here we have a two level multi-level page table, hence we can only create pages of two sizes. Increasing the height of the multi-level page table enables to create pages of more sizes.

Reservation list

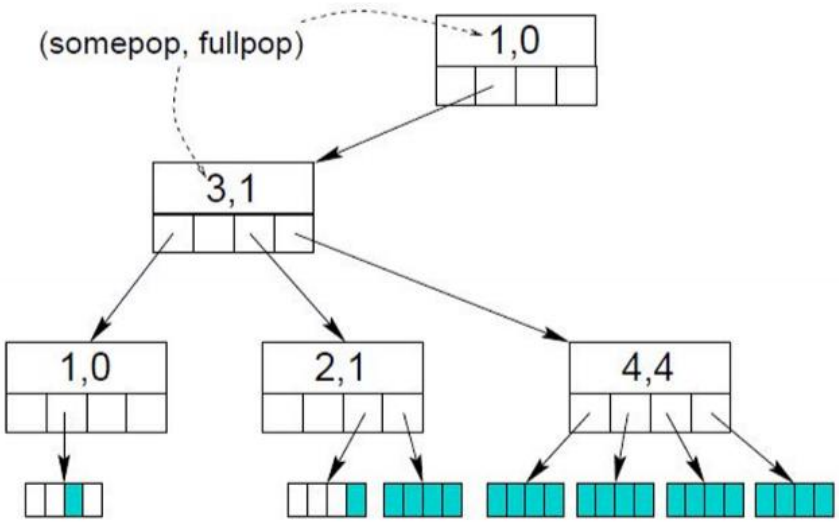
The reservation list is the internal data structure that the kernel uses to track all the allocated pages to processes. The pages are first separated by size, and then sorted in LRU order:



Here the OS supports 3 different sizes. The LRU is important because when it needs more space in memory and hence needs to swap a page to disk, it must know the LRU page as it is the best candidate for least overhead.

Population map

The population map is the data structure that is used to track all the pages allocated to a process. This is used to know whether promotion or demotion is necessary. It is implemented using a Radix Tree:



Each level of the tree represents a page size. The bottom level represents the base pages allocated to that process. The nodes of the tree contain two values, the first represents the number of children, and the second represents the number of children that are “full”. A full child means that the process used all the addresses within the given range. For instance the right most bottom node has all its nodes coloured, meaning the process uses all those addresses withing that range. As we can see, the four children of the node are full, and hence the OS could decide to promote those base pages to a super page.

Promotion and demotion

The OS can promote and demote pages to change their size. This has several advantages.

Promotion helps because it reduces the number of entries in the TLB needed for that address range, and hence it will improve the performance.

Demotion helps if there is a lot of internal fragmentation in a super page. Also, this is sometimes necessary as we want to have a finer grained control/permission on certain parts of a super page, so de split it into several base pages. Also, we sometimes want to swap a huge page to disk, so we first split it into base pages and only send to disk the necessary amount of base pages.

Evaluation

As we can see in the table below, enabling huge pages gives a positive speedup for all the workflows presented, especially the last one where it achieved a speedup of 7.5 !

Bench-mark	Superpage usage				Miss reduc (%)	Speed-up
	8 KB	64 KB	512 KB	4 MB		
Web	30623	5	143	1	16.67	1.019
Image	163	1	17	7	75.00	1.228
Povray	136	6	17	14	97.44	1.042
Linker	6317	12	29	7	85.71	1.326
C4	76	2	9	0	95.65	1.360
Tree	207	6	14	1	97.14	1.503
SP	151	103	15	0	99.55	1.193
FFTW	160	5	7	60	99.59	1.549
Matrix	198	12	5	3	99.47	7.546