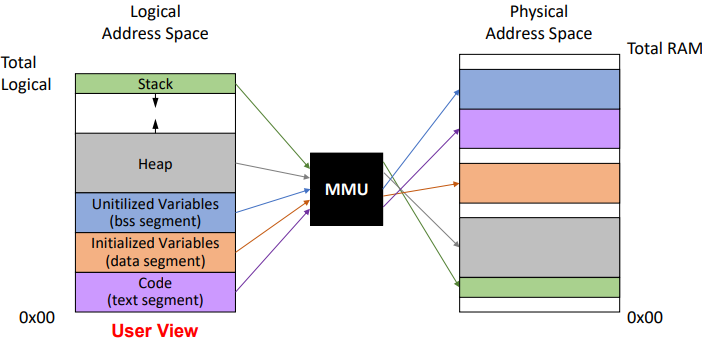
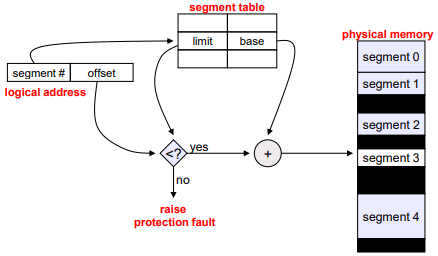
Operating Systems

Paging

Memory Allocation

Memory allocation can be done contiguously or non-contiguously. In contiguous memory allocation, the allocation granularity of a process is its entire logical address space, a program requests enough physical space for all of itself at once and this must be contiguous (all in one piece) in both logical and physical memory. Contiguous memory allocation suffers from issues such as fragmentation, long swap times and long compaction times as previously discussed. Non-contiguous memory allocation is where the logical address space is split into chunks, so again a program will request enough space for all of itself and the logical memory will be contiguous, but there is no requirement that the physical memory must be contiguous. Non-contiguous memory allocation has two key mechanisms, segmentation and paging.

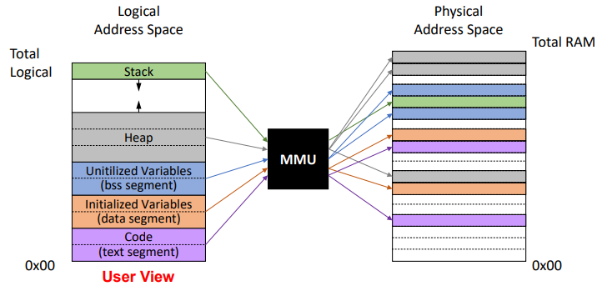
Segmentation

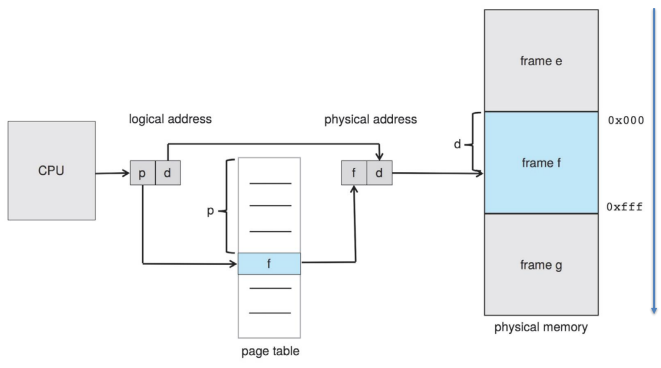
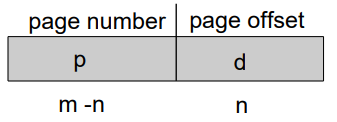
 Segmentation allows us to partition an address space into variable size chunks/units corresponding to logic units such as the stack, heap, data, code, subroutines. The logical address of a segment is structured as a tuple of <segment number, offset>. Segmentation facilitates sharing and reuse of code.

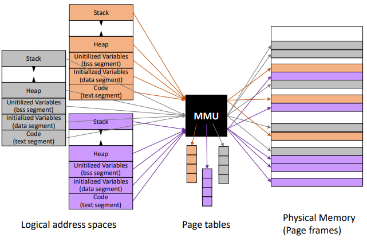
Segmentation requires hardware support in the form of a segment table, this table contains a base limit pair per segment indexed using the segment number. The physical address of a segment is the offset from the logical address + the segment base address from the segment table.

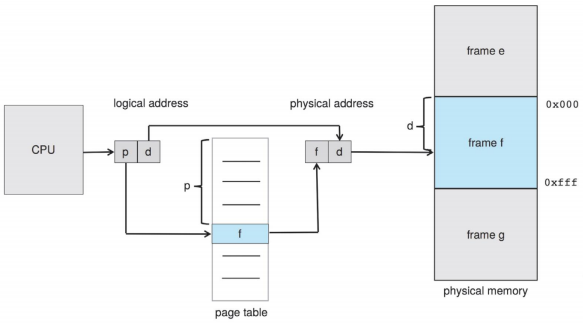
Segmentation allows non-contiguous physical addresses meaning allocated ‘chunks’ are smaller than the entire program address space which in turn reduces fragmentation by exploiting varied sizes of holes. It also enables share though giving multiple process access to the same segment. On the other hand the process view and physical view of memory is **very** different and by implementation process can only access their own memory (even if it overlaps with another process).

Segmentation is rarely used on modern systems.

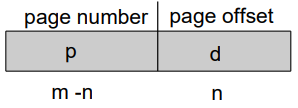
Paging

Paging divides the logical address space into fixed size chunks known as pages. We then map these pages to physical chunks known as frames (the sizes of pages and frames are equal). The logical address is divided into two parts again, the page number which is used as the index into a page table containing the frame’s base address and a page offset which is added to the frame base address to make the physical memory address.

Paging doesn’t suffer from any external fragmentation, but it will suffer from internal fragmentation, the severity of which will depend on the page size, in the worst case the fragmenation will be a whole frame – one byte and on average will be half the frame size. Howeber small frames aren’t always desireable as this makes trnalation more costllly. Again the process and physical views of memory will be very different and by implementation a process can only access its own memory.

Address Translation Scheme

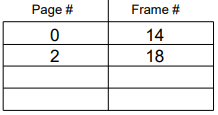
In paging the logicla chunks (pages) map directly to physical chunks (frames) of the exact same size. These chunks are at predefined fixed addresses (both logically and physically).

The logical address is divded into two parts, the Page Number (p) which is the index into a page table containing the base address of the corresponding frames and a Page Offset (d) which is added to the frame base address (once retreived from the page table) to make the finaly physical memory address. (the logical address space is 2m bytes and the page/frame size is 2n bytes).

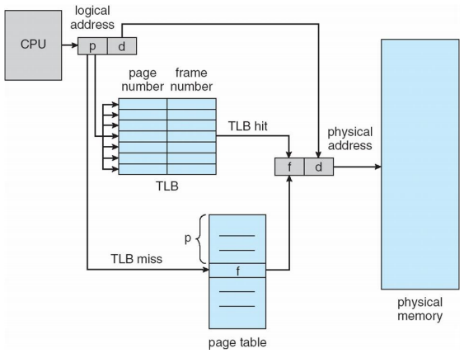
Advantages Of Paging

Paging will create no external fragmentation, however it will create internal fragmentation the degree of which depends on the page size with the worst case being 1 frame – 1 byte and the average case being half a frame. This begs the question of whether small frames are desirable to reduce the internal framgentation however translations may require memory accesses (costly).

Implementation of Page Table

Each process has a page table kept in main memory, it also has stores a page-table base register (PTBR, part of the CPU) to point to the page table and a page-table length register (PTLR, also part or the CPU) to indicate the size of the page table. Every data/code access requires two memory accesses, one to fetch the page table entry (for translation) and another to fetch the actual memory content.

TLB

We want to avoid carrying out two memory accesses as this will result in considderable slow down, to do this we can use translation look-aside buffer (TLBs). TLBs are fast-lookup hardware chaches using associative memroy for parallel searching and having entries of the from <page #, frame #>, they are also typically small (64-1024 entries).

If a translation exists in the TLB (a hit) then we use the data from the TLB with no addictional cost, if it doesn exist in the TLB (a miss) then we must fetch the translation from memory. We also maintin translations for subsequent memory accesses meaning replacement policies must be condidred and some entries can be wired down for permanent fast access.

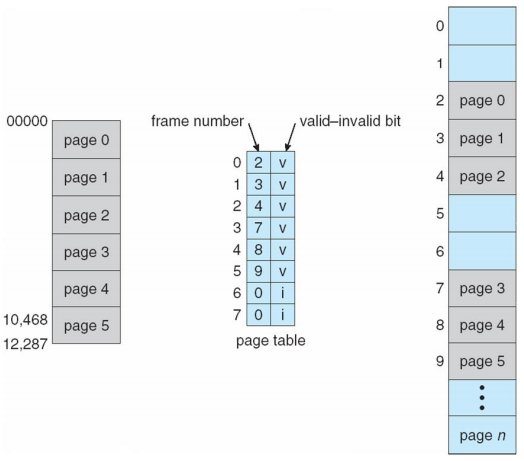
Effective Access Time (EAT) With TLB

The memory access time is the time the CPU waits to access main memory directly. The hit ratio α is the percentage of times that a page number is found in the TLB. The miss ratio is 1 – α. To calculate the EAT we weight the hit ratio with its memory access time and the miss ratio with its memory access time and sum the two.

Consider α = 80%, with 100ns for memory acces, the EAT = 0.8 \* 100 + 0.2\*200 = 120ns

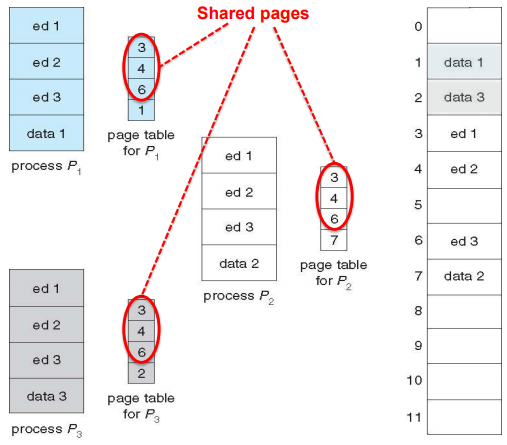
Consider α = 99%, with 100ns for memory acces, the EAT = 0.99 \* 100 + 0.01\*200 = 101ns

Memory Protection

In order to achieve memory protection the page table entries (PTEs) contain more information than just the page number and frame number. We can include protection bits in each PTE that represent:

* What type of access is allowed (read-only, read-write, execute-only)
* Does a translation exist (valid indicates that the associated page is in the process’ logical address space, invalid indicates that the page is not)
* Etc…

Any violations will result in a trap to the OS kernel.

Shared Pages

We can use paging to share both code and data. One copy of read-only (reentrant) code can be shared among processes, such as different processes sharing the same library code, the process will still have per-process data. Read-write data pages can also be shared among processes for communication. This is completely different from multithreaded processes where different threads share the entire address space.

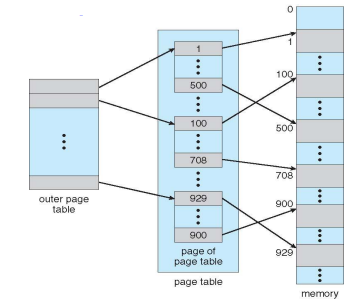
We can also keep code and data private where each process keeps a separate copy of the code and/or/data.

Pages for the private/shared code and data can appear anywhere in the logical address space.

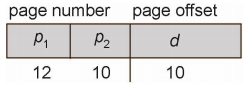
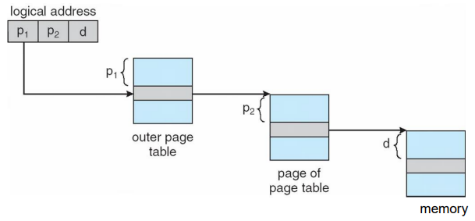
Structure of the Page Table

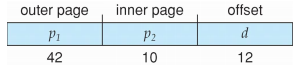
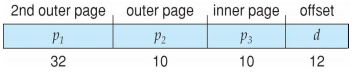
Page tables can get **huge**, consider a 32-bit logical address space with pages each being of size 4kB (212), the page table would have 1 million entries (232-12 = 220) with each entry being 4 bytes meaingin 4MB of memoyr space will be required which will need to be contiguous.

Alteratively we could use other construcstiosn such as **Hierarchical**, **Hashed** or **Inverted** page tables.

Hierarchical Page Tables

We break up the entire logical address space inot multiple page tables, then construct another talbe that referes to each of those page tables.

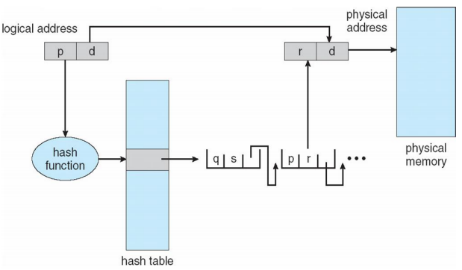
 Here is an example of two-level paging. The logical address is 32-bits with 1K page size divided inot a pagenumber (22 bits) and a page offset (10 bits). The page tablie is paged, the page number if further divided into a 12-bit outer page number and 10-bit inner page number.

For a a 64-bit logical address space a two-level paging scheme won’t be enough, if the page size is 4kB (212), a single-level page table will have 252 entries, a two level scheme with an inner page table having 210 entries, the outer page table would have 242 entries (4,398,046,511,104). If a three level scheme is use (we add a second outer page table) the outer page table will still have 232 entries (4,924,967,296).

On 64-bit machines, the logical address space is typically < 264 for practical resaons (there are no machines with 18 zetta bytes) but 4-level and 5-level page tables o exists.

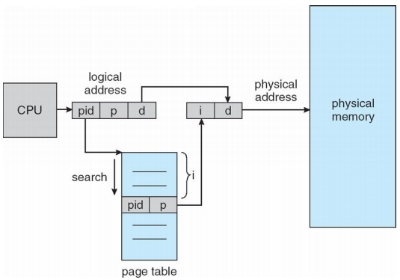
Hashed Page Tables

For hased page tables the logical page number is hashed inot a page table index, each entry chains elements hashing to the same location and each element contains:

* Logical page number
* Value of the mapped page frame
* Pointer to the next element

The logical page numbers are compared searcing for a match, if one is found the corresponding physical frame is extracted.

A variation for 64-bit addresss called clustered page tables exists, this is similar to hased, but each entry refers to several pages (16 rather than 1).

Inverted Page Table

Rather than each process having a page table and keeping track of all possible logical pages, we track all the physical pages. We have one entry for each physical page of memory with each entry consitsing of the virtual address of the page and information about the porcess that owns the page (for protection). This decreases the memory needed to store trnaslation but increases the time needed to search the table, we use a hash table to limit the search to one/few page-table entries.

Shared memory with inverted page tables needs OS intervention.

Why Page Tables

Page tables allow:

* Inter-process memory protection (the address XYZ for process A is different than the same address for process B)
* Protect code from being rewritten (code pages can be read-only meaning a bad pointer can’t change the program code)
* Detect null pointer dereferencing at runtime (the first page of the logical address space, as such references to address 0 cause a trap to the OS)
* Reuce memory usage (shared libraries) (have one copy of a library in physical memory, not per process, all the page table entires to the library point to the same st of physical frames)
* Generalise the use of “shared memory” (regions of two process’ address spaces map to the same frames)

Copy-on-write (CoW), instead of copying all the pages on fork, we create a shared mapping of the parent pages in the child address space, these shared mappings will be read-only for both processes and when either process writes, a fault occurs and the OS ‘splits’ the page.

Memory-mapped files, instead of using open, write, close, we map a file inot a region of the logical address space (e.g. into region with base X), accessing the logical address ‘X+N’ refers to offset ‘N’ in the file, initially all the pages in the mapped region are marked as invalid. The OS reads a page from the file whenever an invalid page is accessed and writes a page to file when evicted from physical memory.