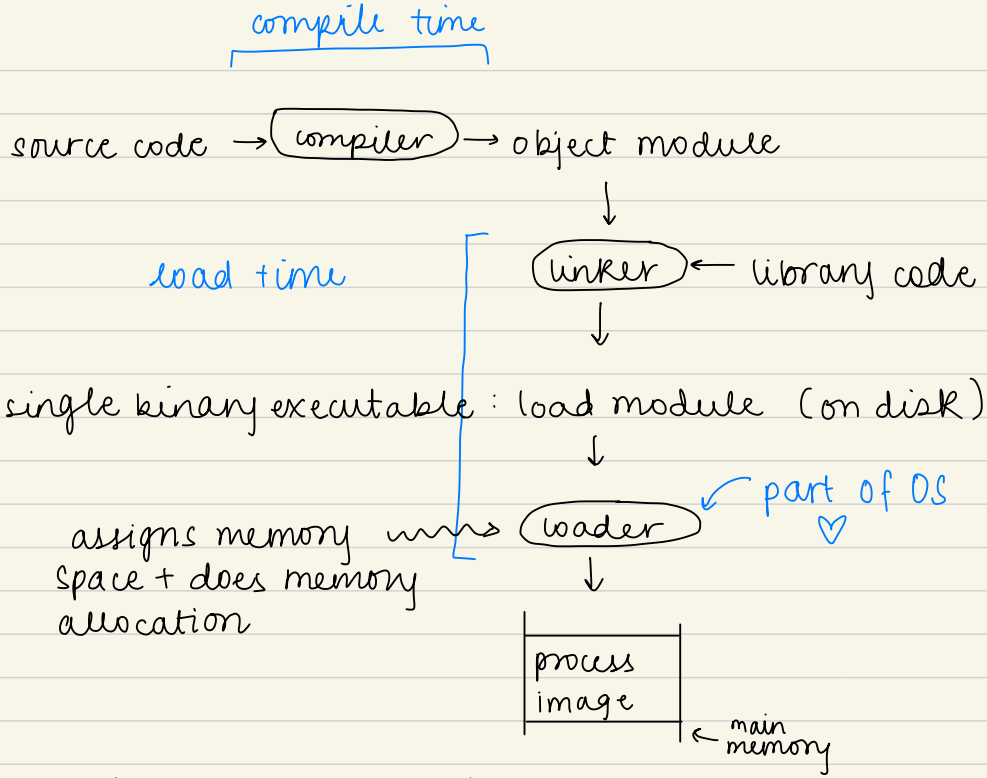
Bind Code and Data to Memory

- create process ⟶ add it to memory for execution
- multiprogramming is executing multiple processes concurrently
  - ↓
  - multiple processes need to be loaded into memory to ensure a reasonable supply of ready processes to consume available CPU time
    - ↳ when more processes are in the memory, the chance of getting a process in the ready state to occupy the CPU is higher
- memory allocation for a process image / logical view of memory / logical address space

meaning! → address reference generated when executing an instruction always refers to this space!



execution of process involves accessing instructions from data and memory

always logical address

eg. LOAD X
↳ address reference of data to be loaded.

during exe.

CPU must deal w/ memory references generated,

① Branch instructions contain an address to reference the instruction to be executed next

② Data reference instructions contain the address of the byte or word of data referenced

↑
these addresses may not be used directly to access physical memory, depending on when address binding is performed.

Address binding of instructions and data to memory addresses can happen at 3 different stages

1. Compile time : if memory location for the process is known at compile time, then absolute code can be generated. If at a later time the location is changed, the code will have to be recompiled

2. Load time : if we don't known the memory address at compile time, compiler generates relocatable code. binding is performed by loader into absolute address

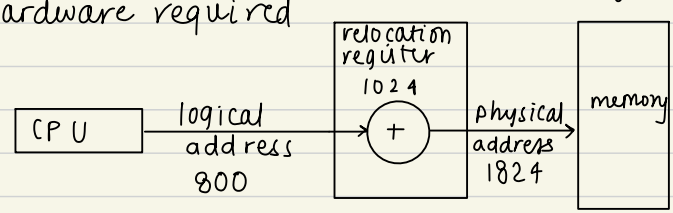
address = offset from beginning

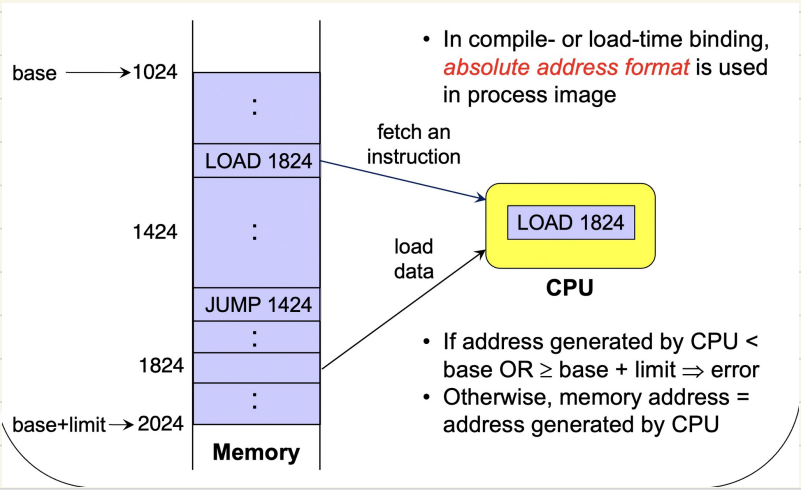
OS approved!

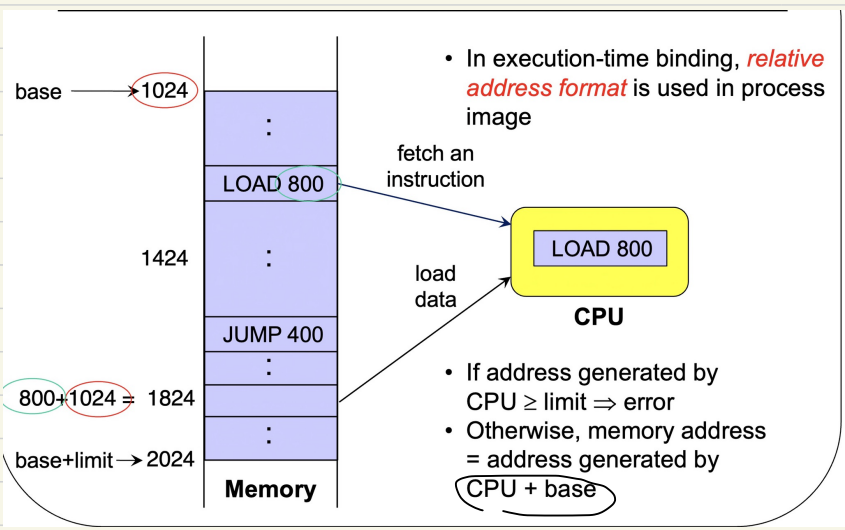
3. Execution Time : If the process can be moved during its execution from one memory segment to another, then binding has to be delayed to run time.

special hardware required

base + limit







↳ translation done by MMU

base, limit registers ⟶ supported by HW

- compile + load time binding, absolute address is used directly to access the memory.
- So the binding of logical address space to physical memory is static. ⇒
- once process image is loaded, it won't move

why execution binding?

- in multiprogramming systems, the available memory is generally shared among a number of processes. Typically it is not possible for the programmer to know which other programs will also reside in physical memory at the time of his program's execution.

- we want the following functionality

1. memory compaction
2. swap in and out process
3. provide virtual memory support

Logical v/s Physical Address Space

- address space : all addresses accessible by a process
- logical address : address used in the code, generated by CPU when executing instruction
- physical address : address used to access physical memory, seen by memory unit

translation may be required

only for execution time

- logical address space can be viewed as linear, or one-dimensional, continuous address space consisting of a sequence of bytes
- for execution time binding, logical address space always starts at zero

Process Memory Allocation

- Contiguous
  - fixed partitioning
  - dynamic partitioning
- non-contiguous
  - process logical address space is scattered over different regions in physical memory

Contiguous Memory Allocation

- the logical address space of a process remains contiguous in physical memory

Fixed Partioning

- memory is partitioned into regions with fixed boundaries
- pre-determine number and sizes of partition
- when there's a request, find a large enough partition to assign to that process.
- problem :

① fixed number of partitions ⇒ fixed number of processes that can run simultaneously ⇒ degree of multiprogramming is fixed.

② need to know memory requirement of process beforehand to be able to decide size of the partition but that is almost impossible to know

③ no matter how small the process is, it still occupies a whole section
↑
internal fragmentation

Dynamic Partitioning

- hole : block of available memory ; initially theres one big hole
  - holes of various size are scattered throughout the memory
- when process arrives, it is allocated memory from a hole large enough to accomodate it
- OS maintains info about
  - → allocated partitions
  - → holes

how to satisfy request of size n from a list of free holes

1. First-fit
2. Best-fit : smallest hole big enough to fit
3. Worst-fit : biggest hole

problem :

→ total unallocated memory space is enough to satisfy a request but can't because it is not contiguous.

↑ external fragmentation ⟶ can reduce by compaction

- Compaction : shuffle memory contents to place free locations together

  possible only if re-locatable address format is used in process image + binding is done during execution

Paging

- memory space allocated to a process can be noncontiguous, process is allocated physical memory wherever it is available

- physical memory is divided into frames
  - fixed size blocks
    - bytes, a power of 2, b/w 512, 8192

* logical address is always contiguous.

- divide logical memory into blocks of the same size called pages
  - → basically process is split into pages

- THERES 1 logical memory space per process and 1 physical memory space for all processes

(page : frame) given a process needing n frames ⇒ n pages
- find n free frames, load program onto it

→ set up a page table to translate logical to physical addresses

- external fragmentation is eliminated
- interanal fragmentation may occur when last page cant occupy the whole block

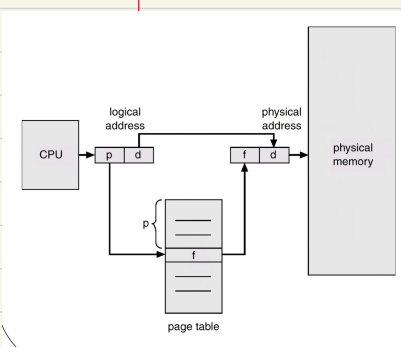
page size = frame size

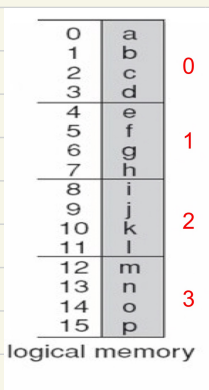
no of pages = $\frac{\text{logical memory size}}{\text{page size}}$

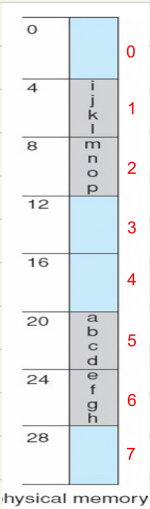
no of frames = $\frac{\text{physical memory size}}{\text{page size}}$

page table :

page #	frame #
0	5
1	6
2	1
3	2



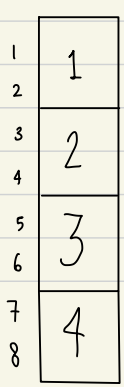




logical address contains

page number (p) → index in page table entry

page offset (d) → combined with frame number to determine physical memory address that is sent to the memory unit



8 bytes ← size of item

2^3 bytes

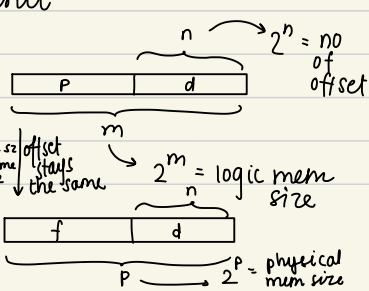
3 bits to rep

no of pages = 8/2 ← size of page

offset per page = 2 / unit ← here, 1 byte

no of offset bits = $\log_2 2 = 1$

no of page bits = 3 - 1 = 2



also saved into PCB during context switch

cant be stored in reg as it is of variable size

Implementation of Page Table

- page table is kept in physical memory (kernel memory)

system register

needed for OS to be able to find table in memory

- page-table base register (PTBR) points to the page table (for each process)
- page-table length register (PTLR) indicates size of the page table

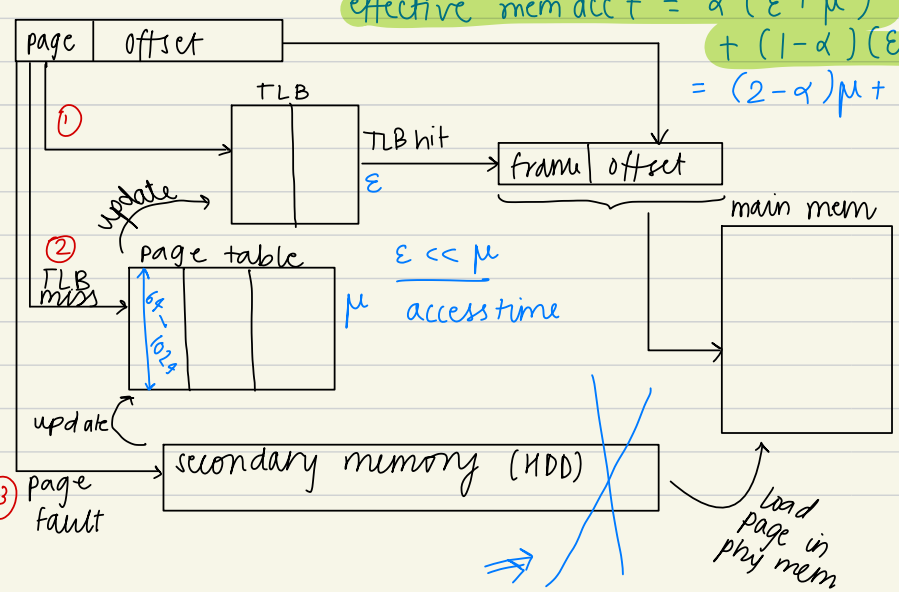
- every data/instruction access requires two memory accesses : 1st for PCB
  2nd for data/instruction

↑

so effective memory access time = 2μ (μ = time mem access) can be reduced by using fast-lookup hardware cache called associative registers or translation look-aside buffers (TLBs)

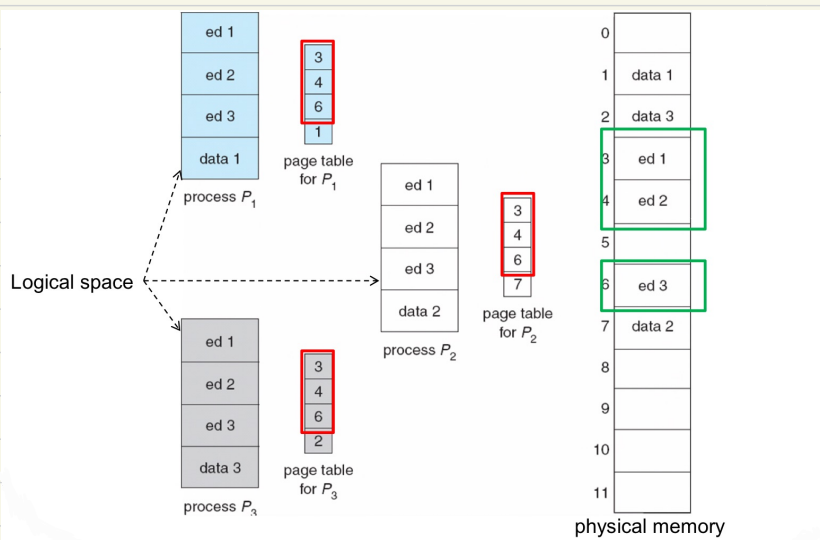
- cant store all, only frequently accessed frames

effective mem acc t = $\alpha(\varepsilon+\mu) + (1-\alpha)(\varepsilon+\mu+\mu) = (2-\alpha)\mu+\varepsilon$

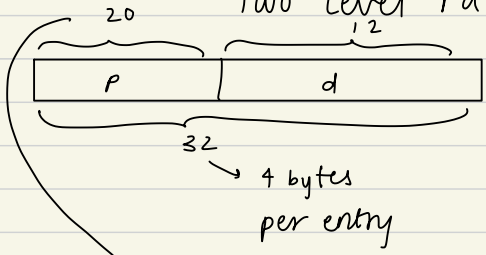


Shared Pages

- if same processes use the same read-only (reentrant) code shared among processes (ie. text editors, compilers, windows system)
- it can be shared by making the page table for all processes having the same frame for the shared pages
- the shared code must appear in the same location in the logical address space of all processes
- each process has its own copy of private code and data whose pages can appear anywhere in the logical address space



Two-Level Page-Table Scheme



$4K \rightarrow 4 \cdot 2^{10} = 2^{12}$

offset bits = 12

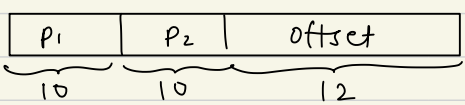
$\Rightarrow 2^{20}$ pages $\Longrightarrow$ $2^{20} \times 4$ bytes for whole table (4MB) ↑ crazy!!

allocating contiguously is a problem ←

a large table is divided up to be easier to allocate in physical memory, w/ small increase in effective access time

2 lvl paging : page table is paged ⇒ it is also split up

eg (same numbers)

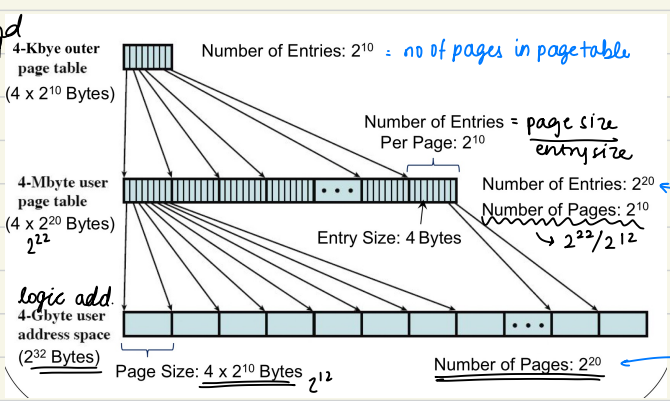


first lvl size / page size ← p_1 → index into outer page table

page size / entry size ← p_2 → displacement within page of inner table

so second level confused

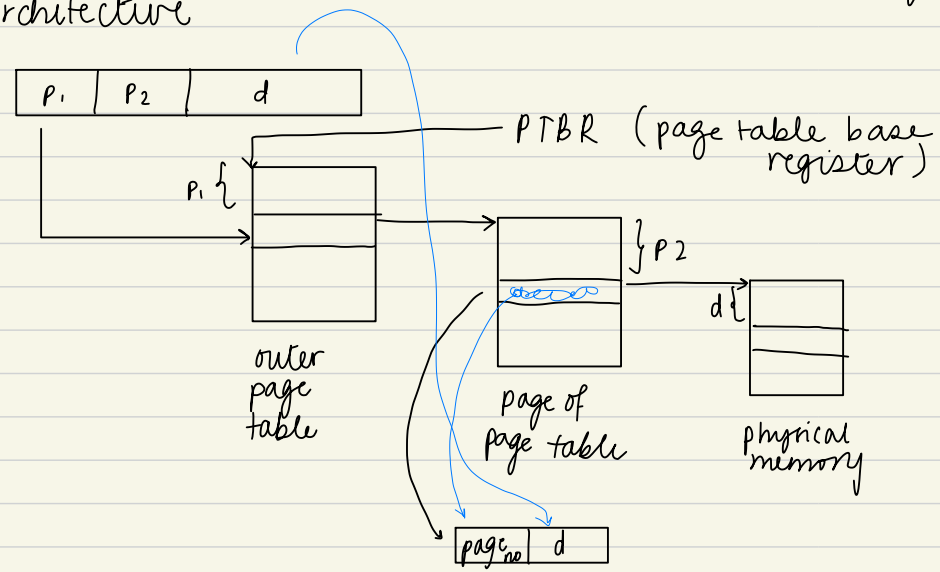
first lvl.



need to keep track of which page of page table is in which frame

address translation scheme for 2 level 32-bit paging architecture

logical address



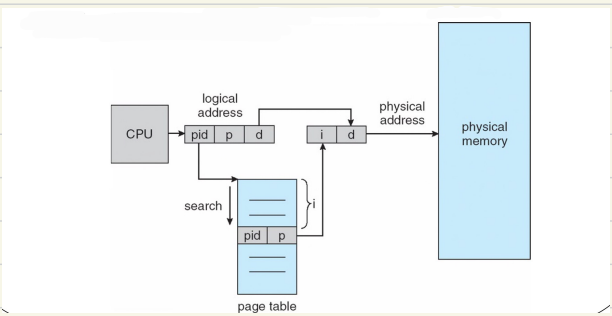
Inverted Page Table

- ∵ page table size ∝ logical address space size
- alternative : One table , 1 entry per physical frame
  - ↳ < process-id , page no >

inverted page table

logical address : < process-id, page no, offset >

- ordered by frame number

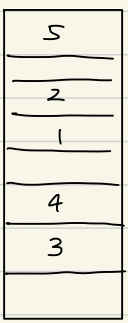


- disadvatage : cost : given page no its quick to find here we have to search whole table to find page and id cause it's arranged w/ frame in mind.

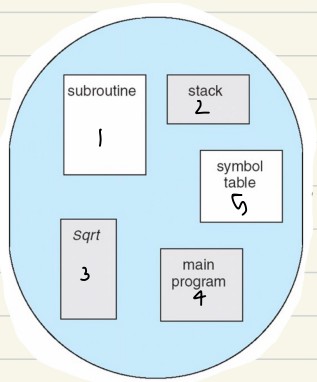
noncontiguous → Segmentation

'segment' → logical unit

logical address space becomes a collection of segments



physical memory



each segment has a size associated with it, can be variable.

segments, like pages, don't have to be allocated contiguously

advantages

1. simplifies handling of dynamic ds which can be assigned its own segment
   as segment (ds) grows, the OS allocates more space in memory or if unavailable, swaps it out and swaps in the enlarged segment at the next opp.

2. programs can be altered and recompiled independently
   ↳ which have their own segment

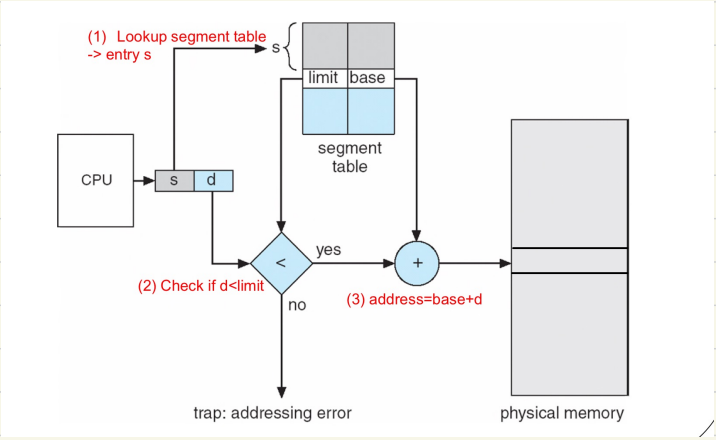
3. lends itself to sharing among process

4. lends itself to protection

segment has a segment no & offset
↳ < segment-no , offset >
↓ in table ↳ in segment

- segment table → each entry has
  limit - length of segment
  base - starting physical address

STBR → mem location of table
STLR indicates number of segments



- dynamic storage-allocation → best / first fit
- external fragmentation ∵ when process leaves, there are holes in mem of diff sizes
  ∵ segmented code, holes are smaller.

summary.

1. process image = logical address space
2. contiguous allocation → fixed partition IF
   dynamic partion EF
3. non-contiguous allocation → paging IF
   segmentation EF