Memory Management

MMU : h/w feature for mem. manage.
h/w detect memory violatiobs. OS takes action accordingly.
OS: Sets up MMU for proc., then schedules process
Compiler : Generates object code for a particular OS + MMU architecture.

Linking and Loading:

1. **Static linking:** All obj. code combined at link time and big obj. file is created
2. **Static loading:** All the code is loaded in memory at time of exec()
3. **Problem :** Big **executable file** need to load functions even if they do not execute.
   -need to be executable and unnecessary memory allocation done to unused code
4. **Solution :** Dynamic linking and Dynamic loading

Dynamic Linking:

Static linker :

Links
- function code to function calls
- references to global variables with "extern" declarations

Dynamic linker :
- does not combine function code with object code file
- instead introduces "stub" /placeholder code that is indirect reference to actual code
- at time of loading/executing prog in mem, the "**link-loader**" will pick relvent code from library machine code file

Function code is refernced ni PLT
PLT : Procedure Linking Table
used to call externel procedures/fun whose addr is to resolced by dynamic linker at run time.

Loader :
- loads prog in mem
- part of exec() code
- need to understand format of executable file (ELF)

Dynamic loading :
loads part of elf file only if needed during execution
delayed load
more sophiscate mem manage by OS
Dynamic linking necessary deamnds **advanced tyoe of loader** that understands it
- called as "linker-loader"

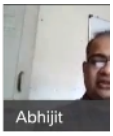
Continous memory management

Continous mem mange :
entire process is hosted in continous chunk of memory
Mem divided in 2 partitions :
1. OS : high memory as interr vector map to this
2. Processes

Problems faced by OS:

find continous chunk for proc to fork
diff. proc are of diff size // need of size parameter in PCB
free mem after use
maintain list of free areas // can be done in array or linked list

Variable partition scheme



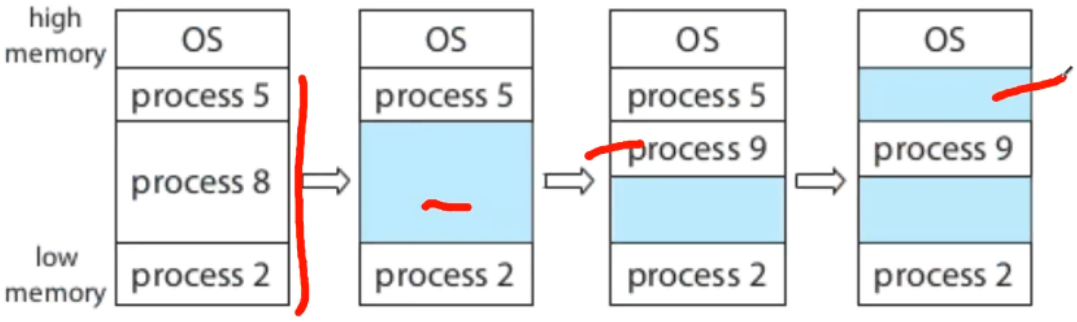


Figure 9.7 Variable partition.

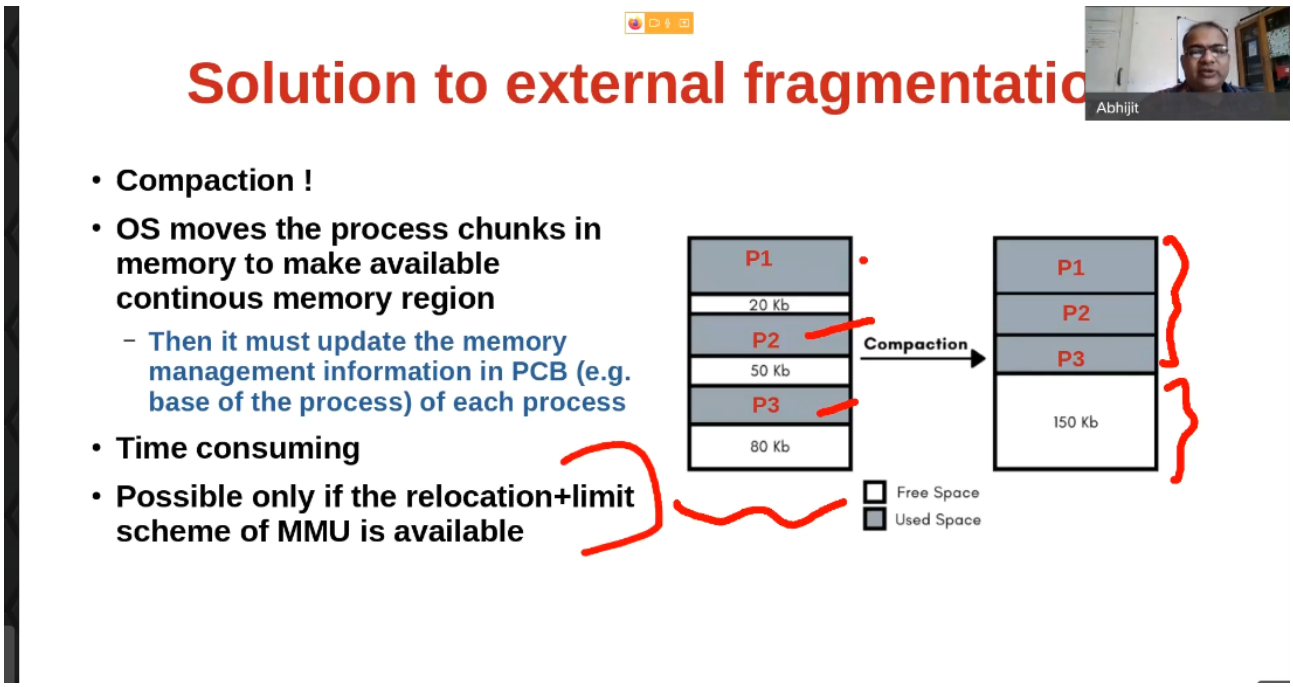
How to find holes in this :

Best fit : smallest hole larger than process
Worst fit : largest hole
First fit : first hole larger than the proc.

Problems : external fragmentation:

if 30k, 30k, 40k holes are there but we can't assign 50k proc to it.

Solution : Compaction



Another solution if external fragmentation :

- fixed partition scheme
- Memory is divided by OS in chunks of equal size

if chunk sze : 50k

proc 40k = 1chunk
proc 140k = 3chunk
proc 60k = 2chunk

This may lead to internal fragmentation : internal wastage in chunks allocated

Fixed Partition scheme:

OS have to track :

if partion is free or used by which proc
- partition tracked by using bitmap.list of num
- PCB will contain list of parttion used by it

Solution ti internal fragmentation :

- reduce size of fixed partion
- if too small more overhead for OS in allocation-deallocation

generally 4kb

Paging

Extended verison of Fix size part.

partition = Page

- Process = logical byte seq. **Divided in "page" size**

- mem divide in equal sized page **frames**

Process need not be continous in RAM

Different page size chunk of proc in any page frame

Page table maps pages into frames

Logical addr : pagenumber + offset

pagenumber : gives frame number when as index in page table

offset : + frame gives physical addr.

Schema :

1. Compiler assumes continous memory . One chunk

genrates machine code

2. h/w MMU : convert logical addr to physical

MMU's job

To translate a logical address generated by the CPU to a physical address:

1. Extract the page number p and use it as an index into the page table.

(Page table location is stored in a hardware register

Also stored in PCB of the process, so that it can be used to load the hardware register on a context switch)

2. Extract the corresponding frame number f from the page table.

3. OS – when proc is created (fork/exec), creates a page table

allocate frame to proc

fill page table entries

In each PCB, maintain

- page table allocation (addr)

- list of pages frames allocated to proc

During context switch of proc, load PTBR using PCB

<u>Global tasks also to be done by OS :</u>

maintain list of all page frame

1. allocated frames
2. free frames (frame table)
3. can be done in linked list
4. innovative data struc

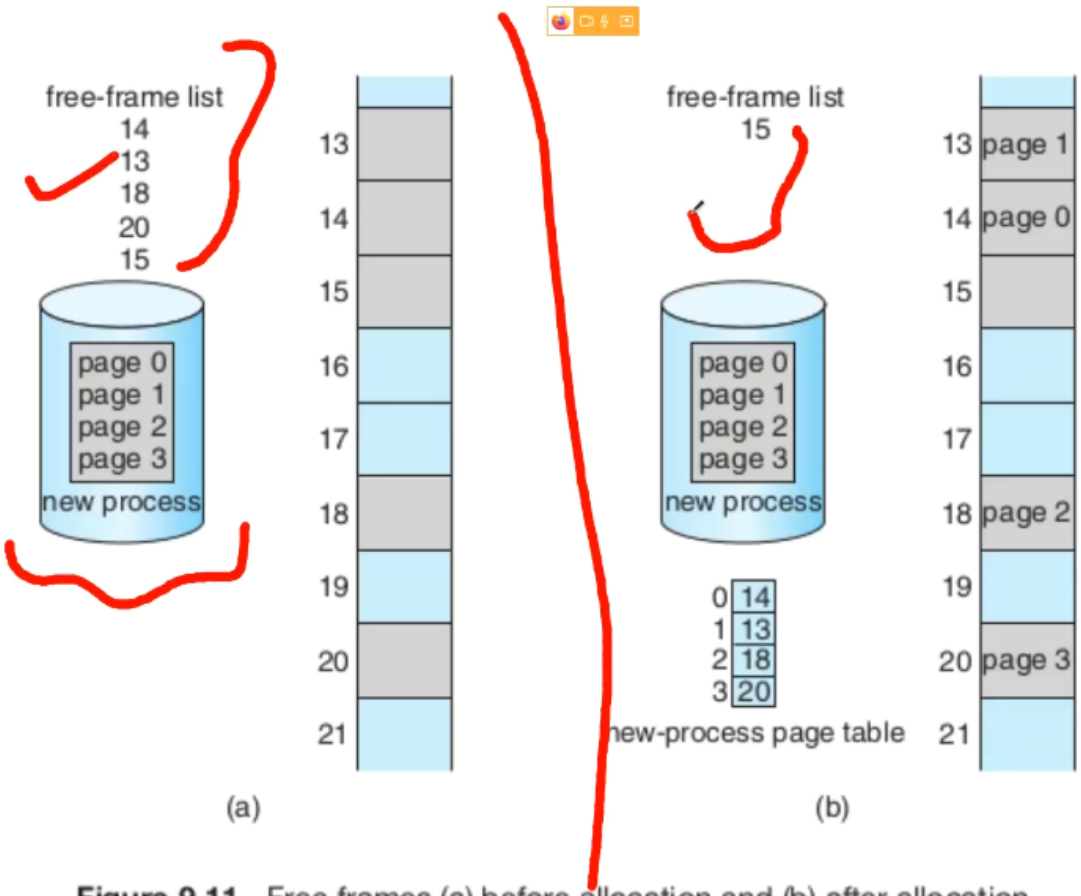


Figure 9.11 Free frames (a) before allocation and (b) after allocation.

Disadavatanges :

- each mem acc. has 2 mem access
  - one for page table one for actual mem loc
  - done as part of execution of instr. in h/w.
  - slow down by 50%

TLB (Translation lookaside buffer):

part of CPU h/w

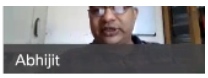
cache of page table entries (subset)

contain both page number and frame number

searched in parallel for page number // constant time

update TLB during each acess to page table by **h/w automatically.** NOT KERNEL.

Speedup due to TLB



- **Hit ratio**
- **Effective memory access time**

  = Hit ratio * 1 memory access time + miss ratio * 2 memory access time
- **Example: memory access time 10ns, hit ratio = 0.8, then**

  effective access time = 0.80 × 10 + 0.20 × 20

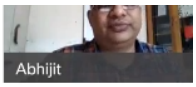
  = 12 nanoseconds

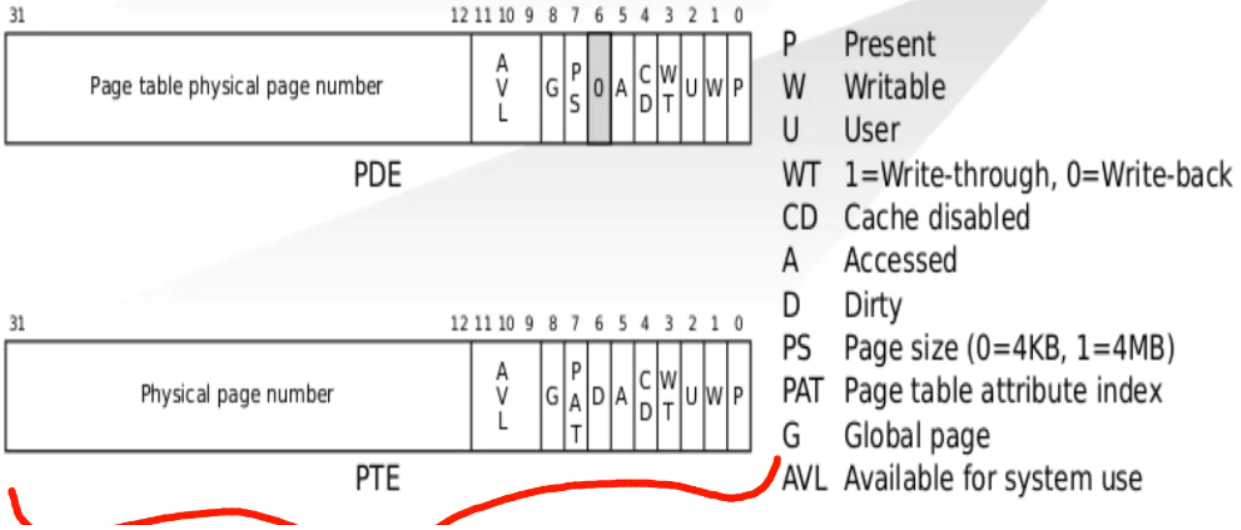
Memory Protection paging:

if a pointer to random addr. If addr is more than file then it is marked invalid, indicating illegal memory addr. Kernel detects this and gives interrupt

in xv6 more such bits are there.

X86 PDE and PTE





Shared pages (Library with paging)

Kernel should understand that part of elf is lib code.

It should allocate only one instance of lib code (Shared library)

All proc should map to the same instanc of lib code

the **lib code should map to same virtual addr.** If they differ mapping will map to diff loc in page table.

Paging : Problem of large page table

if 64 bits

if 20 bit off.

1mb pages

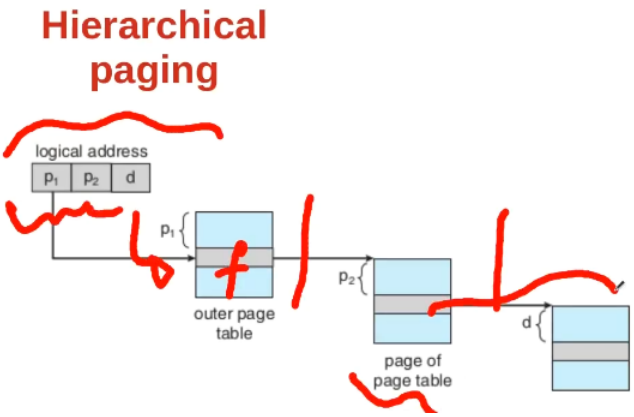
44 bit page number : 2^44 size page table

Too large continous page tables

if page size is increase internal fragmentation

in 32bit also same problem persists.

Hierarical Paging



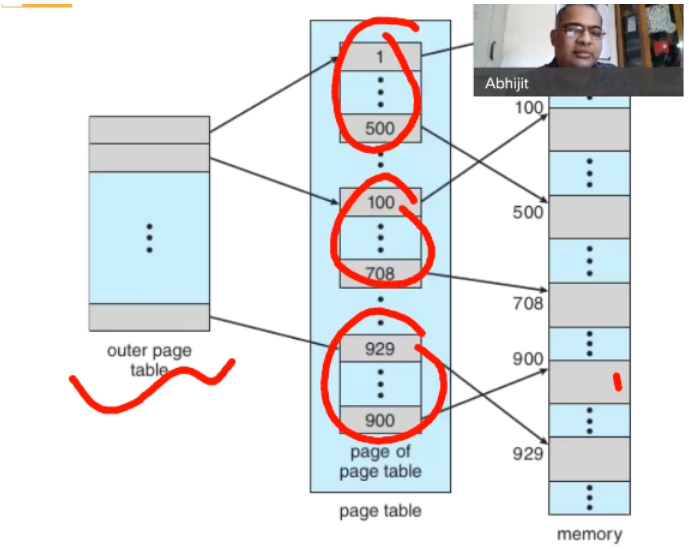


Figure 9.15 A two-level page-table scheme.

outer page	inner page	offset
p_1	p_2	d
42	10	12

p2 p1 off : for 2 level. 3 memory accesses

Can have 3 level also. 4 mem accesses.

Problem :

- too slow: more number of mem. access for each level

- OS data struc also needed in same propor,

Solution:

Hashed Page table

Hashing : reduces search time by using hashing function which gives index in hash table.

May be happen collisiosn

Solved by collision resolution methods

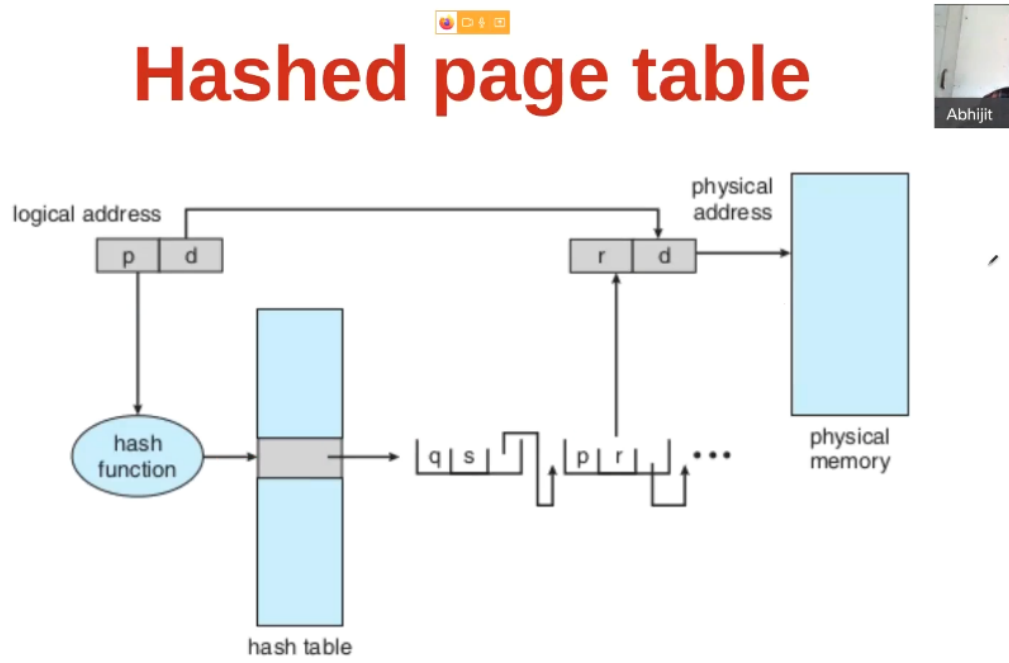


Figure 9.17 Hashed page table.

Very Costly.

Hashing done by cpu.

Hash table in RAM.

If collison then interrupt, then OS will run linear search during collision

Inverted Page table:

Inverted page table

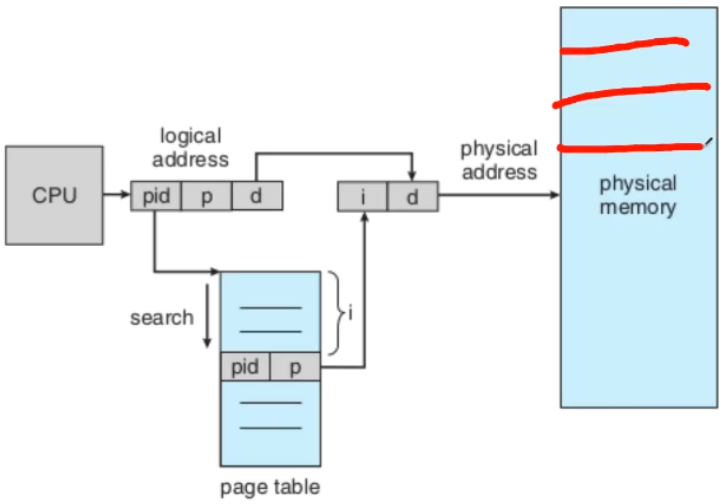


Figure 9.18 Inverted page table.

here it is frame table

here only one process table.

No per process tables . **Saves a lot of memory.**

It should a pid to see which proc does the page belongs to.

get logical addr, pid + p + d

search in page table based on index (page number and pid) -> get frame number

frame + offset : physical addr.

pid : is not generated by CPU. In register

Case study : Oracle Open Solaris

64 bit SPARC processor

uses hashed page table

- one for **kernel**
- one for all **user proc**

hash table entry : base + span(#page)

reduces number of entries required

doesn't have mapping to each pages

Caching is done at 3 levels:

1. **TLB (on CPU)**
2. **TSB (in Memory)**
3. **Page table (in memory)**

- CPU implement tlb that hold translation table entry (TTE) for fast h/w lookup

- cache of TTE reside in in memory translation storage buffer(TSB), which includes an entry per recently accessed page

- when vir. addr. reference ocurs, h/w search TLB f0r translation

- if none , h/w walks though TSB looking for TTE for the vir. addr.

- if found in TSB , CPU copies it in TLB. Mem trans. is done.

- if not in TSB then kernel is **interrupted** to search in hash table

- the kernel then creates TTE from appro. Hash table and **stores it in TSB for automatic loading into TLB by CPU**'s MMU

- finally interr handler return control to Mmu, which completer addr. trans. And retrieves requested byte from main memory.

Xv6 and linux : hierarchical paging

Swapping

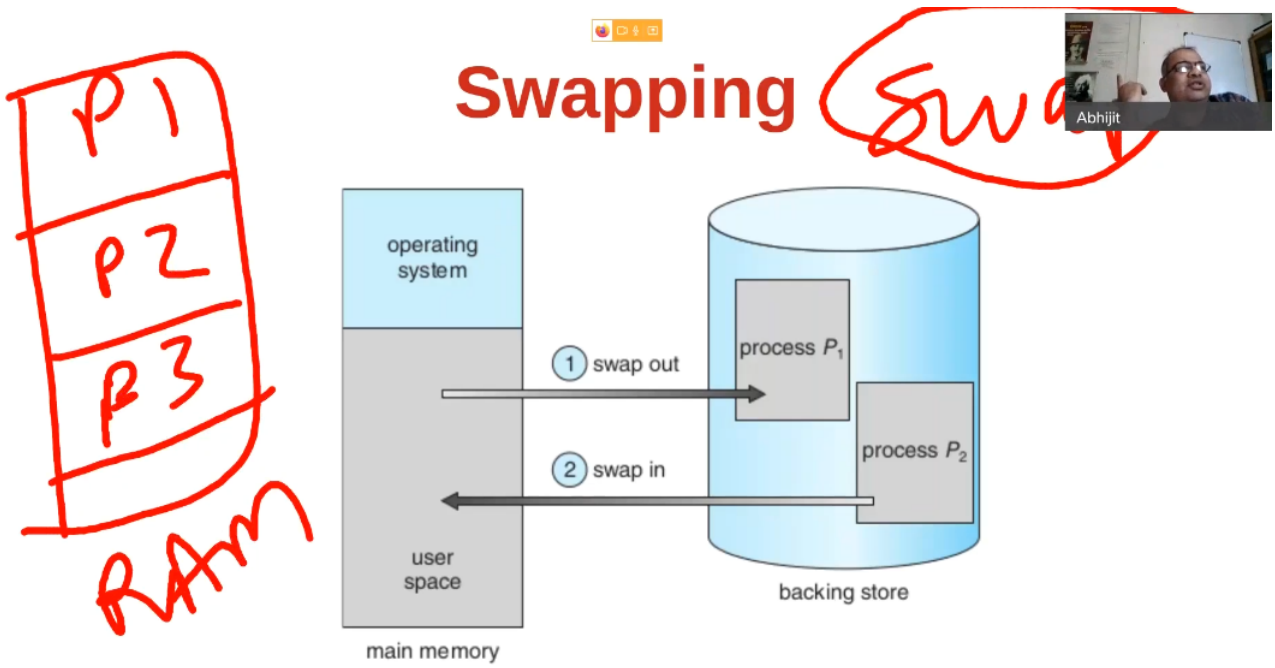


Figure 9.19 Standard swapping of two processes using a disk as a backing store.

if proc. in RAM is not used, then take it in **backing store (swap)**

then again if required we can swap it in again.

Standard Swapping :

- entire proc swapped
- conti. Mem. management

Swapping with paging :

- only some pages are “paged in ” or “paged out”
- # “paging ” =~= swapping

Stack and heaps : have pages also

code and shared lib also