int *y= (int *) — in stack
malloc (sizeof(int)) — in heap.

Date:

sizeof()

→ Routines and macros are utilized for size in malloc instead typing in a number directly.

→ Two types of results of sizeof with variables.

→ The actual size of 'x' is known at run-time

```
int *x = malloc (10* sizeof (int));
printf( "%d \n", sizeof(x));
```

→ 10 × 4 = 40

to print the size of address

4 ✗ size of how big a ptr to integer is

→ The actual size of 'x' is known at compile-time.

```
int x[10];
printf( "%d \n", sizeof(x));
```

CORRECT WAY

40. ✓ size of x[10]

↳ size of x in array.

free()

```
void free(void* ptr)
```

→ free memory region allocated by a call to malloc.

→ Argument

① void *ptr: a pointer to memory block allocated with malloc.

→ Return

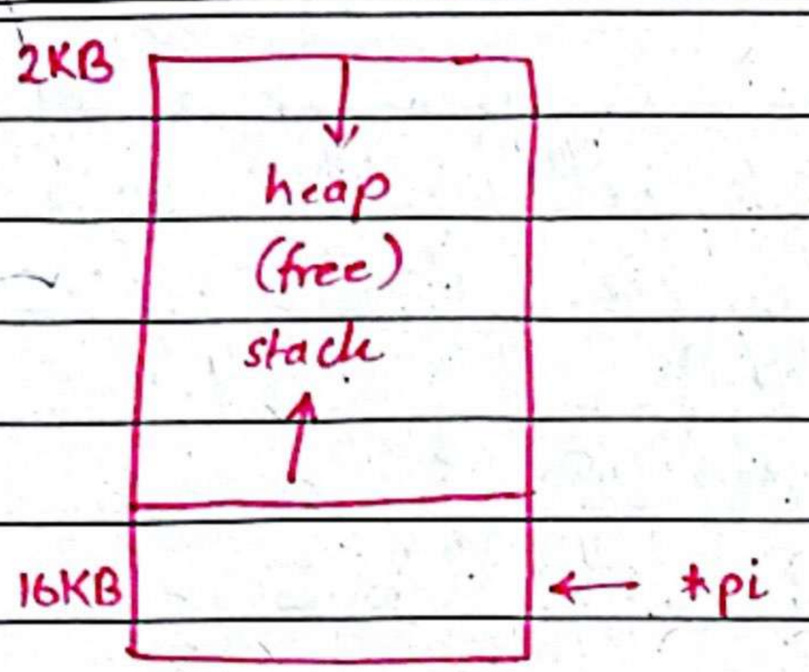
① none.

SP PAPER PRODUCT

* malloc will tell where the code is present

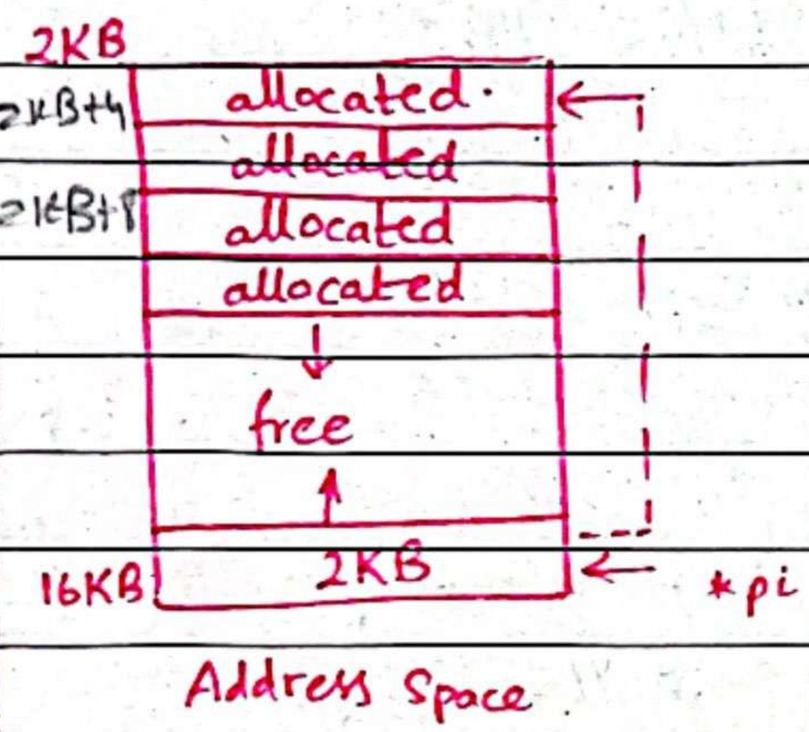
MEMORY ALLOCATING.

Date:



---> pointer

int *pi ; //local variable.



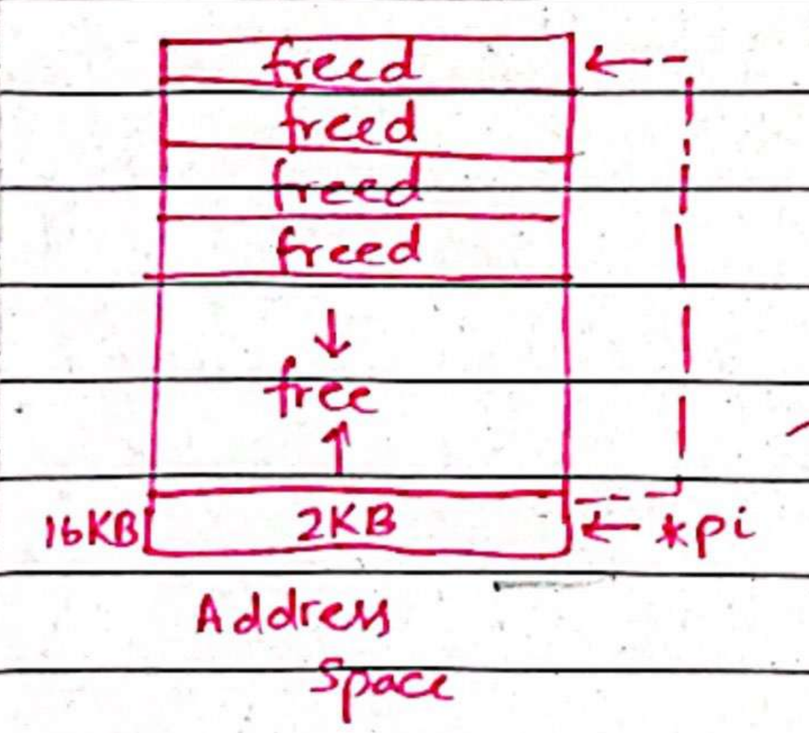
pi = (int *) malloc (sizeof(int) * 4) ;

4 x 4

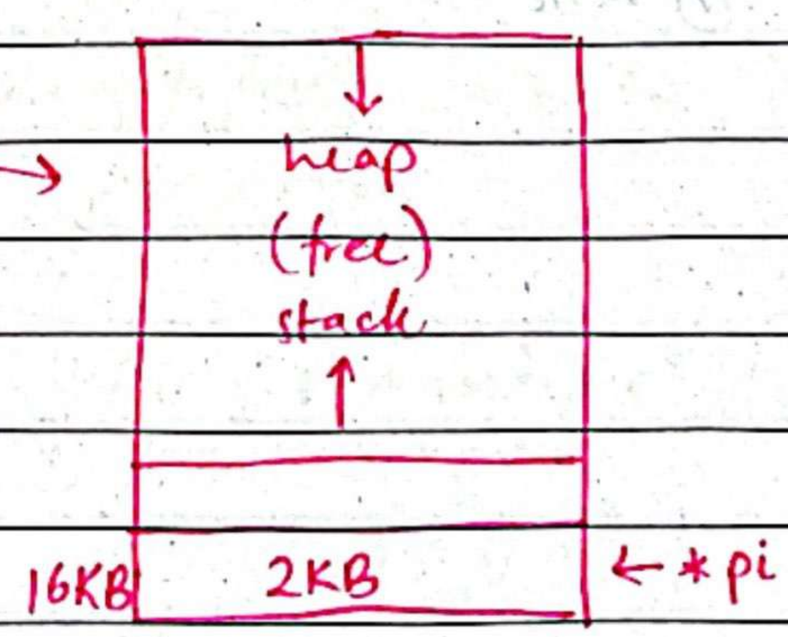
(i.e.

$\underbrace{\text{int *pi = (int *)}}_{\text{stack}}\ \underbrace{\text{malloc (sizeof(int)*4)}}_{\text{heap.}}$

MEMORY FREEING.



free(pi);



SP PAPER PRODUCT

Date:

Forgetting To Allocate Memory

→ Incorrect Code

```
char *src= "hello";   // character string constant.
char *dst;            // unallocated
strcpy(dst,src);      // segfault & die.
```

→ Correct Code.

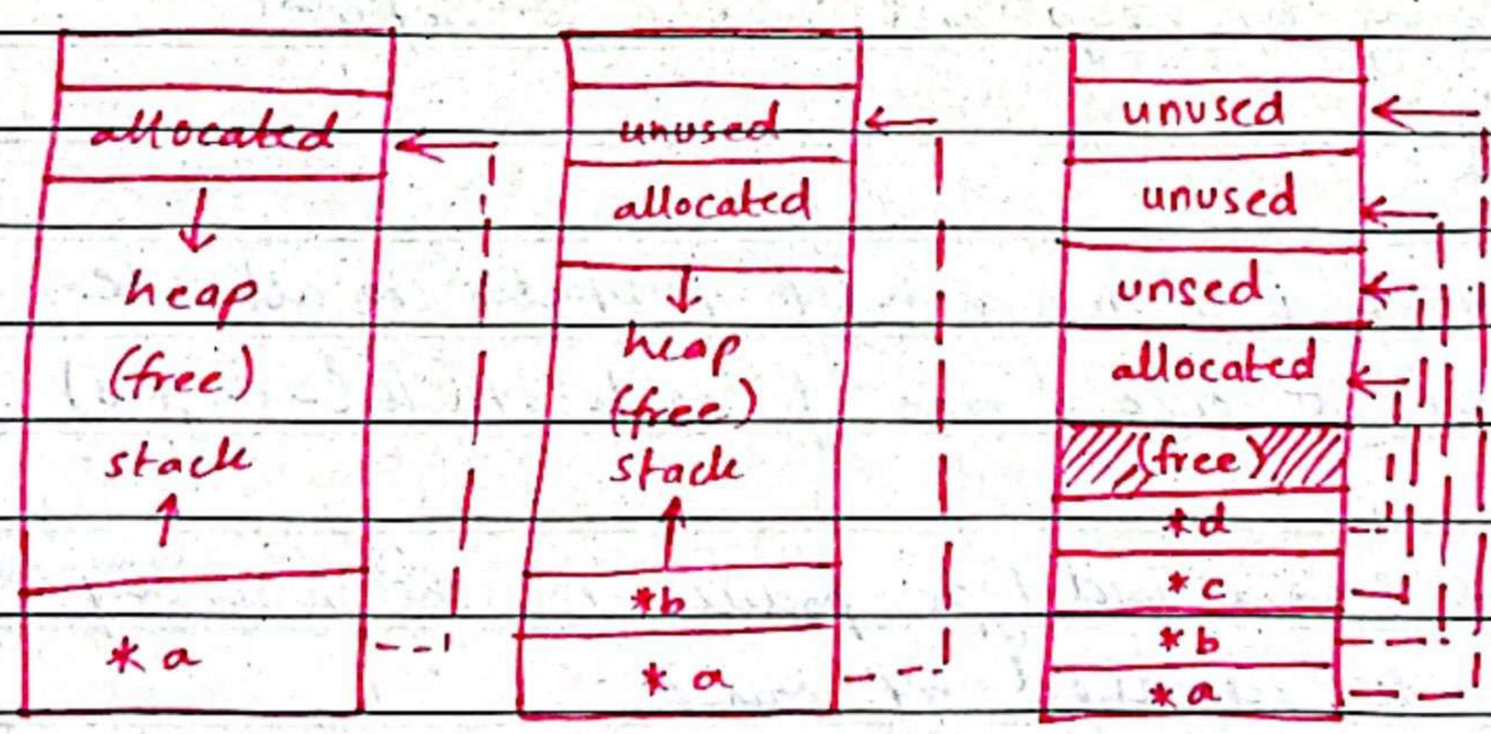
```
char *src="hello";
char *dst (char *) malloc (strlen(src)+1);   //allocated.
strcpy(dst,src);  //work property
```

(termination)

MEMORY LEAK

→ A program runs out of memory and eventually dies

unused : unused, but not freed.



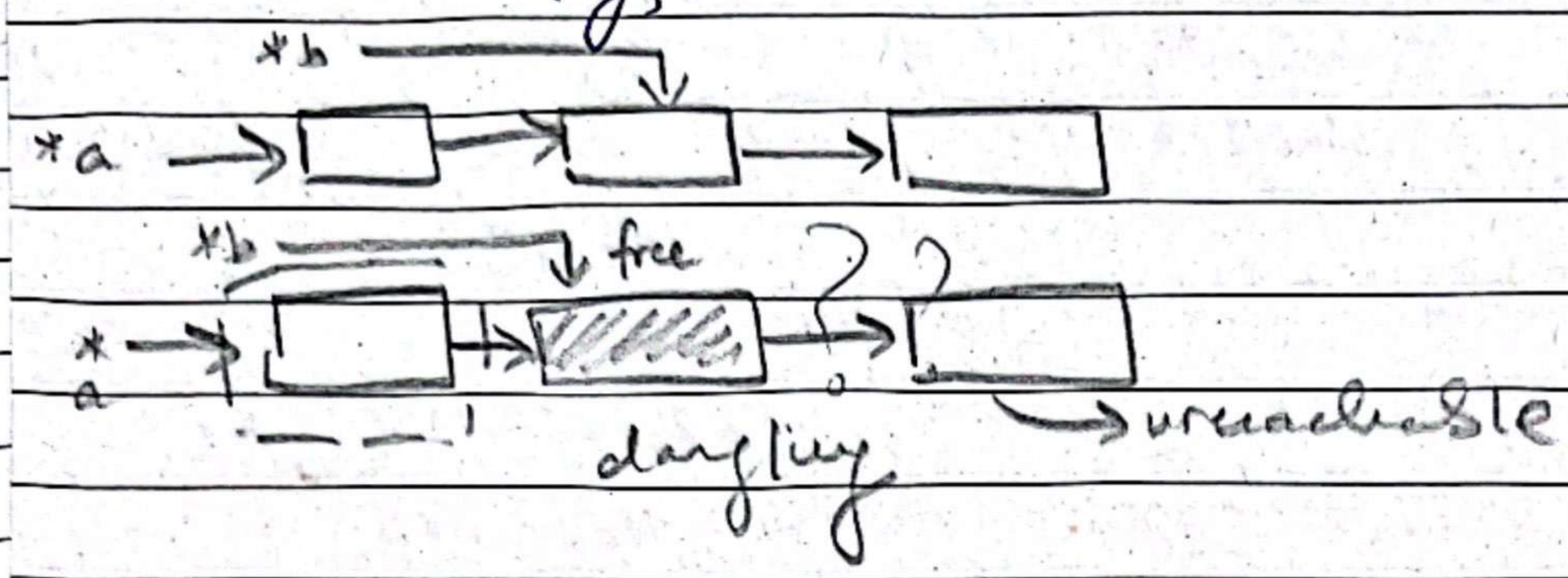
→ RUN OUT OF MEMORY.

SP PAPER PRODUCT

Date:

DANGLING POINTER.

→ Freeing memory before it is finished ~~during~~ using,



calloc( )

void *calloc (size_t num, size_t size)

→ Allocate memory on heap and zeroes it before returning.

→ Argument

① size_t num : number of blocks to allocate.

② size_t size : size of each block (in bytes)

→ Return

① Success : a void type pointer to the memory block allocated by calloc.

② Fail : null pointer

```
arr = (int *) calloc (n, sizeof (int))
arr[0] = 0
arr[1] = 0
arr[2] = 0
```

SP PAPER PRODUCT

Date:

DOUBLE FREE

→ Free memory that was freed already

```
int *x = (int *)malloc(sizeof(int));  //allocated.
free(x);  //free memory
free(x);  //free repeatedly (undefined error)
```

realloc()

```
void *realloc(void *ptr, size_t size)
```

→ Change the size of memory block.

→ A pointer returned by realloc may be either same as ptr or a new

→ Argument

① void *ptr : Pointer to memory block allocated with malloc, calloc or realloc.

② size_t size : new size for memory block (in bytes)

→ Return

① Success: Void type pointer to memory block

② Fail: null pointer.

SYSTEM CALLS

```
int brk(void * addr)
void *sbrk(intptr_t increment);
```

SP PAPER PRODUCT

Date:

→ malloc library call use brk system call

→ brk is called to expand program's break.

→ break. The location of the end of heap in address space.

→ sbrk is an additional call similar with brk.

→ Programers should never directly call either brk or sbrk.

void *mmap (void *ptr, size_t length, int port, int flags, int fd, off_t offset)

→ mmap system call can create an anonymous memory region.

CHAPTER 15: Address Translation.

* address space is parsely populated.

→ Limited Direct Execution:

Let program run directly on the hardware; however, at certain key points in time (say a system call or timer interrupt) arrange so that OS gets involved and makes sure the "right" thing happens.

SP PAPER PRODUCT

Date:

MEMORY VIRTUALIZING WITH EFFICEENCY AND CONTROL.

→ For Memory virtualizing, we want to take something from limited direct execution (LDE) while introducing flexibility.

→ Specifically, we'd like for programs to use their address spaces in whatever way they would like, thus making the system easier to program.

→ In memory virtualizing, efficiency & control are attained by hardware support.

ADDRESS TRANSLATION → hardware present in MMU.

* Hardware transforms virtual address into physical address, creating beautiful illusion.
  → the desired information is stored in physical address.

→ The OS must get involved at key points to set up hardware.

What assumptions you'll make to create a framework for virtualisation?

① User's address space must be placed contiguously in physical memory

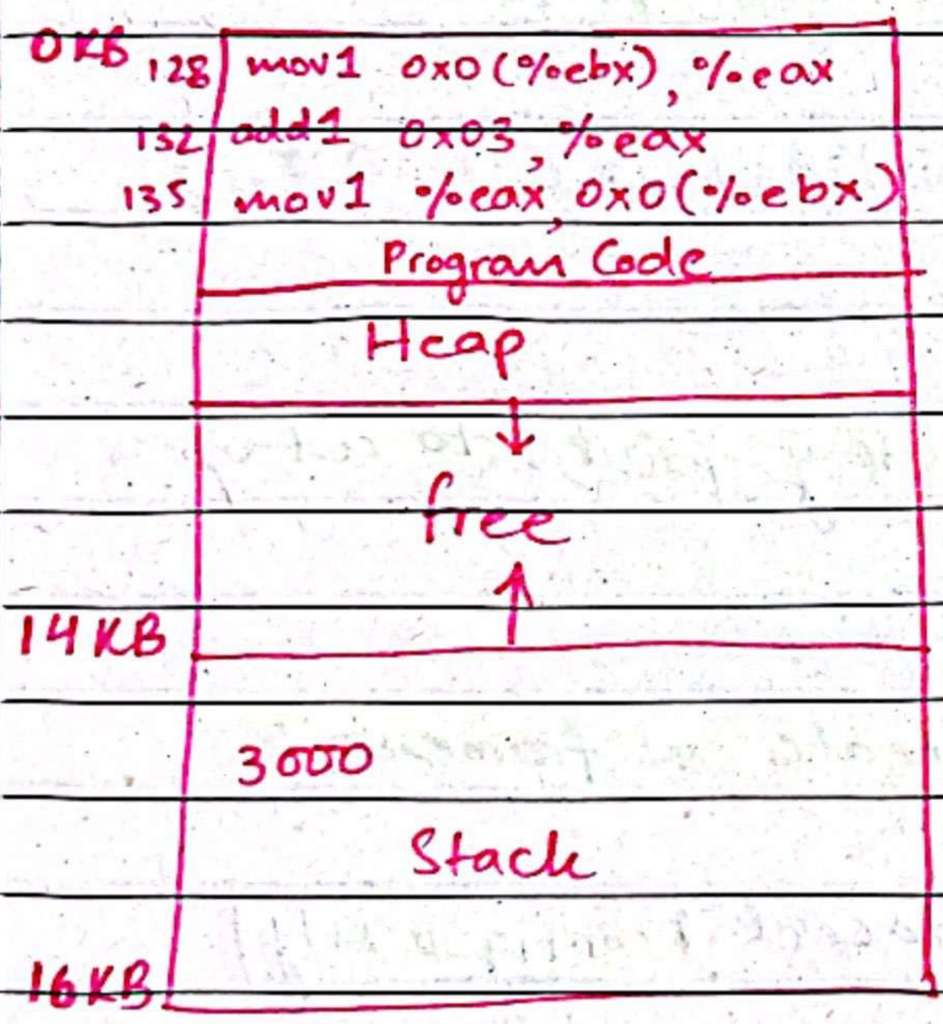
Date:

(2) The size of address space is not too big; specifically it is less than size of physical memory.

(3) Finally, we will also assume that each address space is exactly the same size.

C-language Code. ⟹ Objdump (in Linux)

```
void func()                    ⟹ 128: movl 0x0(%ebx), %eax
  int x;                          132: addl $0x03, %eax
  ...                             135: movl %eax, 0x0(%ebx)
  x = x + 3;
```



→ Note: Code is present in code space

→ Note: Data is present in stack space

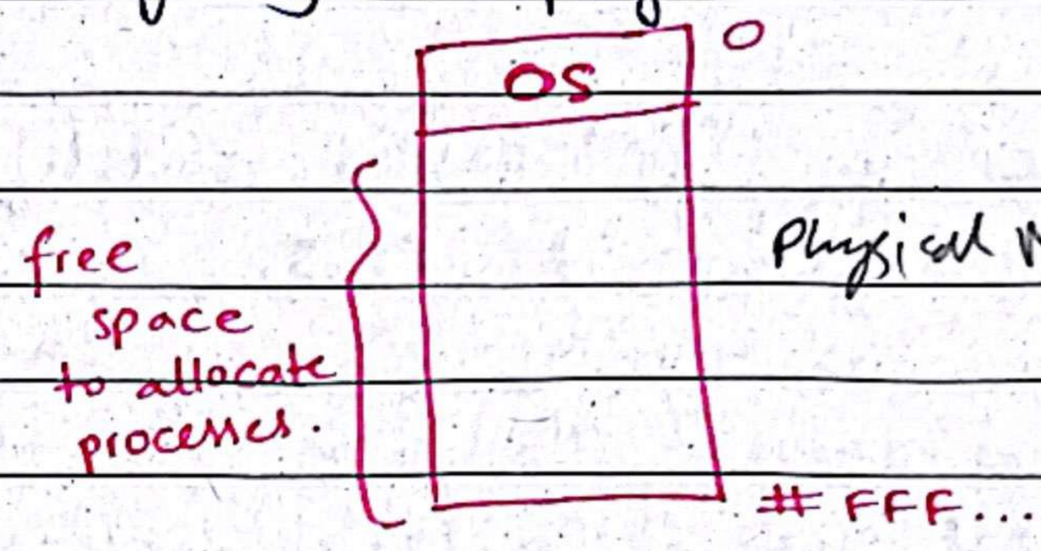
→ Note: address starts at 0 and ends at 16KB.

SP PAPER PRODUCT

In Physical Memory, the OS places the process anywhere in RAM not at 0KB.
But in virtualisation process is kept at 0KB.

Date:

→ OS always take O(1) in physical memory.
→ OS is put in the begining on physical memory at address 0.



→ The OS wants to place the process somewhere else in physical memory, not at address 0
→ Why?
* because the OS itself is kept at address 0.

→ When a program starts running, the OS decides where in physical memory a process should be loaded.
→ Set base register a value.
physical address = virtual address + base.

→ every virtual address must not be greater than bound. and negative.
$0 \leq$ virtual address $<$ bounds.
→ base + virtual address.

SP PAPER PRODUCT

Date:

→ We need 2 hardware registers within each CPU : ① base & ② bound registers.

→ Whenever you do context switch, process A saves bound & base to PCB & accesses process's B base & bound → Process Control Block (PCB)

→ Segmentation fault
① if you access space above bound or below base. (outside address space).
② code rewrite.

128: movl 0x0(%ebx), %eax.

→ fetch instruction at address 128
32896 = 128 + 32KB (base)
→ execute this instruction.
→ load from address 15KB
47KB = 15KB + 32KB (base).

SP PAPER PRODUCT

Date:

① physical address ⇒ virtual address + base

② physical address = base + offset.

→ The base & bounds register are part of Memory Management Unit (MMU) of CPU.

* If one CPU execute one instruction only one pair of base & bounds is enough.

HARDWARE REQUIREMENTS:

① Base/Bounds Registers.
→ need pair of registers per CPU to support address translation & bounds check.

② Ability to translate virtual address & check if within bounds
→ circuitry to do translations & check limits

③ Priviledged instruction (s) to update base/bounds.
→ OS must be able to set these values before letting a user program run.

④ Priviledged instruction (s) to register exception handlers
→ OS must be able to tell hardware what code to run if exception occurs.

⑤ Ability to raise exceptions.
→ when processes try to access priviledged instructions or out-of-bounds memory.

SP PAPER PRODUCT

Date:

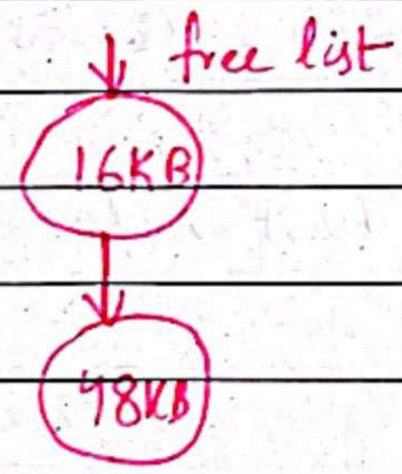
OS REQUIREMENTS.

→ Memory Management (Need to allocate memory for new processes; Reclaim memory from terminated processes; Generally manage memory via free list).

→ Base/Bounds Management. (Must set base/bounds properly upon context switch)

→ Exception handling (code to run when exceptions arise; likely action is to terminate offending process)

* The OS must find a room for a new address space.
→ freelist: A list of the range of physical memory which are not in use.



→ The OS must take action to implement base-and-bounds approach.

① When a process starts running (finding a space for address space in physical memory)

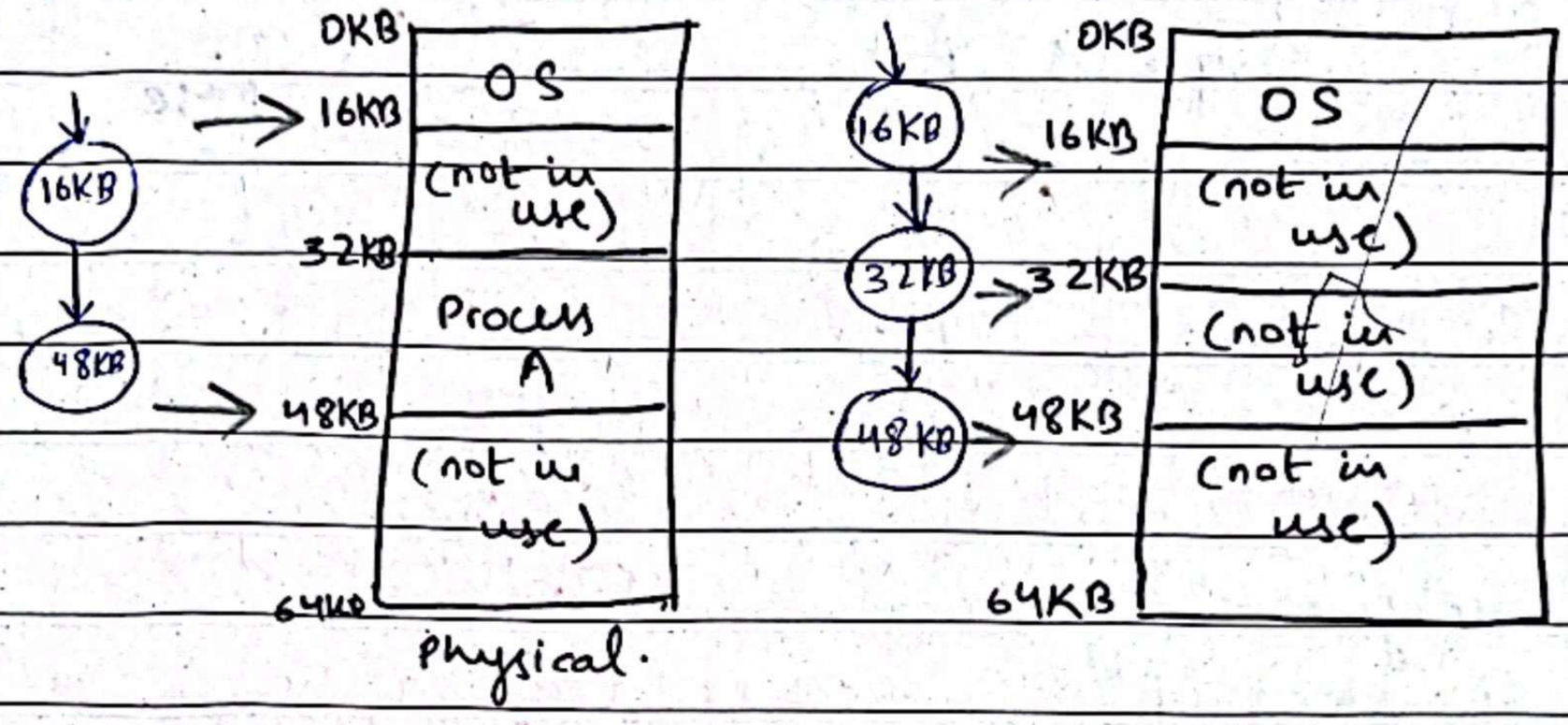
② " " process is terminated. (reclaiming memory for use).

③ " " context switch occurs. (saving & storing base-and-bounds pairs)

SP PAPER PRODUCT

→ Free Up Space: The OS must put memory back on free list.

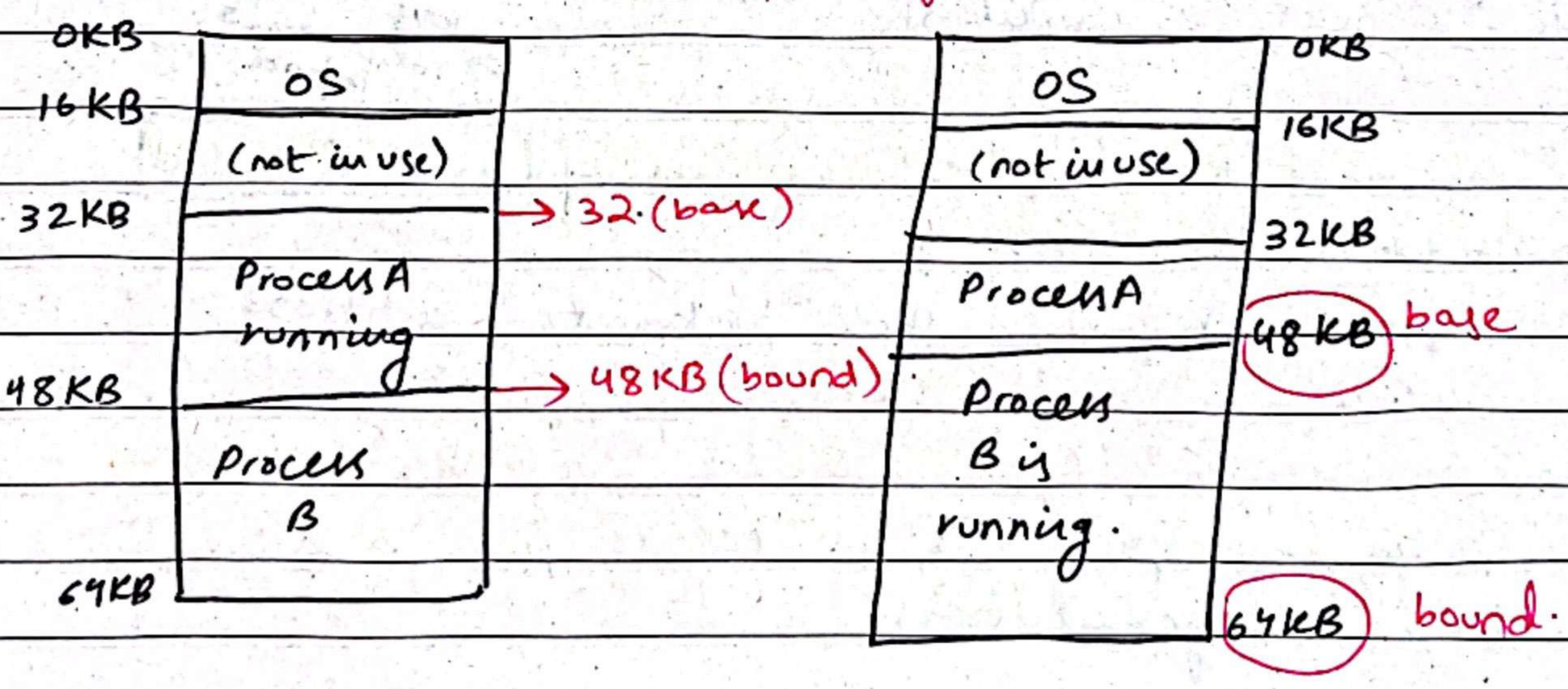
② When process is terminated.



* when process A is terminated free list is updated

③ When Context Switch Occurs.

Context Switch: The OS must save & restore the base and bounds pair in the process structure / process control block (PCB). ① base/bound reg ② Page table.



SP PAPER PRODUCT

* Segmentation vs Paging
→ Segmentation divides memory into variable-sized segments representing divisions of program like code, stack & heap. Paging divides memory into fixed-size pages

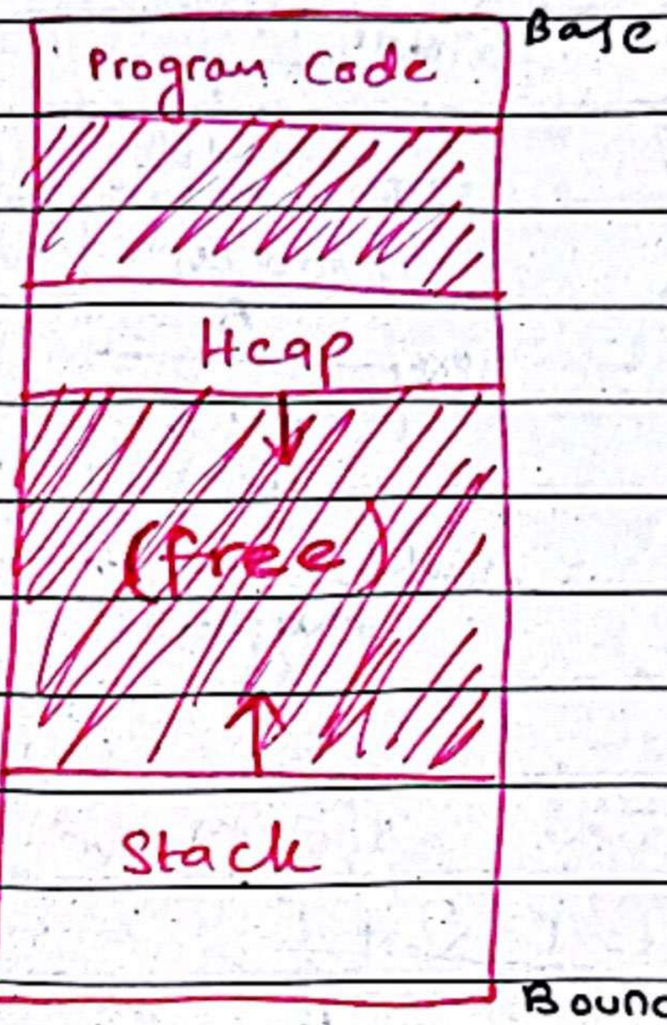
Q) How does OS use the base and bounds registers for address translation in segmentation?
→ OS uses base register to translate logical address to physical address by adding it to offset. The bound register is used to check that the offset is within bounds.

Date:

CHAPTER 16: Segmentation.

→ Virtual Memory comprises of three segments

① Code ② Heap ③ Stack.



→ A segment is simply a contiguous portion of address space of particular length.

→ The "free" is not an illusion, it does take up physical memory.

→ Memory can not be allocated if process demands more & less is available

→ This implementation requires a pair of [base & bound] for address translation. (key components of address)

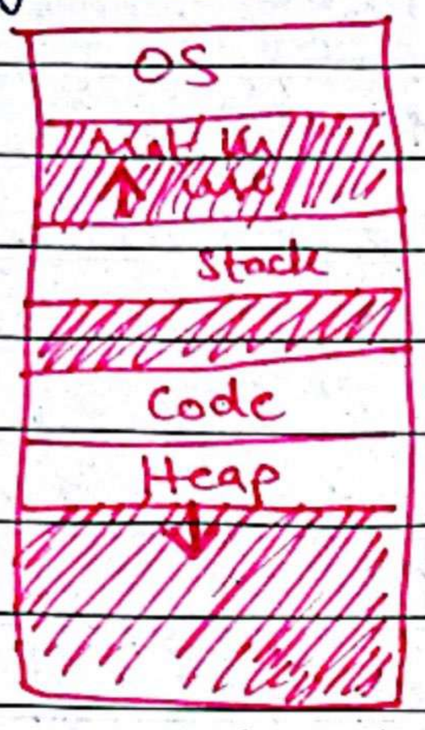
→ each process requires three fundamental entities:
Code ① Program Counter.
Memory ② Base and the ③ Bound register.

→ Again, a lot of space is wasted as free space ...
so, let's do Segmentation.

SP PAPER PRODUCT

Date:

Segmentation



Segment	Base	Size
Code.	32K	2K
Heap	34K	2K
Stack	28K	2K

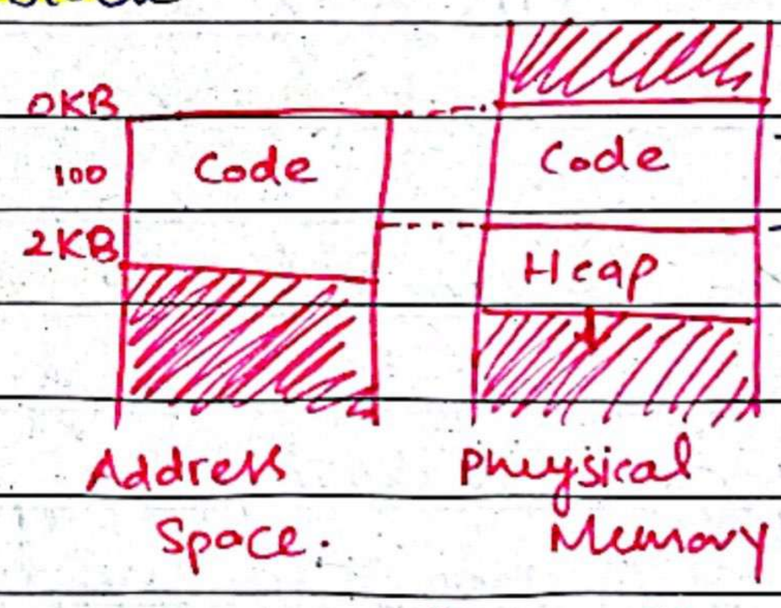
Therefore, 3 segments require 3 (base, size) pairs.

Segmentation fault

① memory access out of your address space
② Modify code block
③ if you try to access header.

→ The term segmentation fault or violation arises from memory access on an illegal address or overwriting something in Code block.

Address Translation



→ Base: 32KB = 32768
→ offset: 100
→ Physical Address
= Base + offset
= 32K + 100
= 32768 + 100
= 32868.

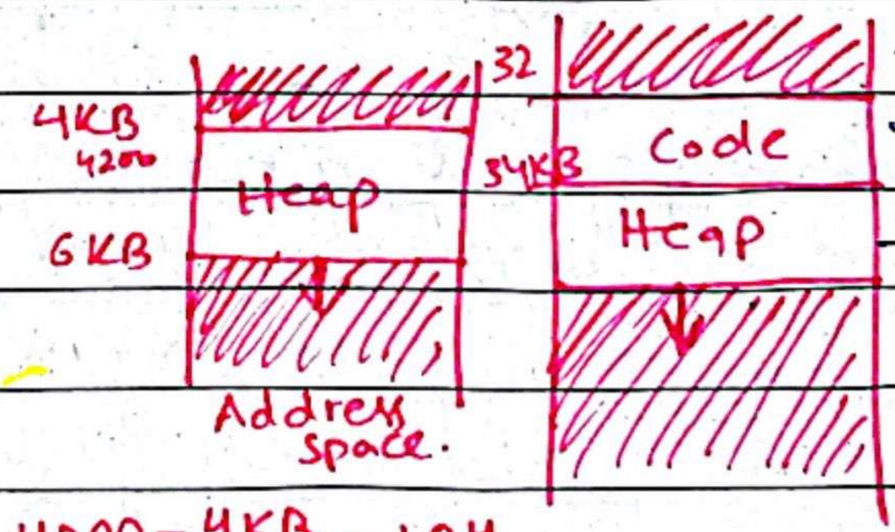
CODE SEGMENT

100 = 0KB + offset
offset = 100

HEAP SEGMENT

Specific data is present at address 4200.

Address = Base + Offset
4200 = 4KB + offset
offset = 4200 − 4KB
= 104.



→ Base: 34KB = 3481
→ offset: 104
→ 104 + 34K
= 104 + 34816
= 34920

4200 − 4KB = 104

SP PAPER PRODUCT

Date:

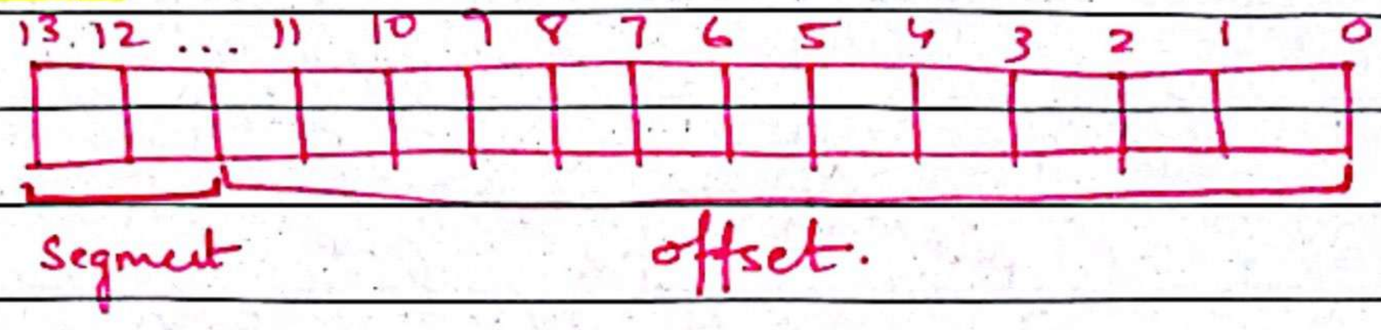
Refering to a Segment

Segment	bits
Code	00
Heap	01
Stack	10

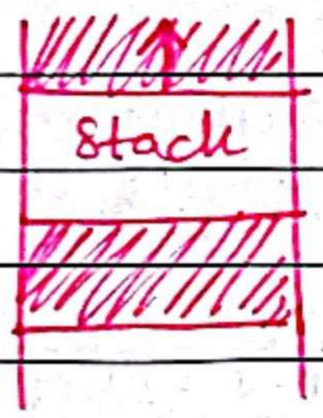
→ virtual Address Space is 16KB

→ 16KB = 16,384 bytes = 2^{14}

→ Therefore, for a 16KB address space, you need 14 bits of address.



→ Refering to Stack Segment



→ Stack grows backwards.

→ Extra hardware support is need.

① The hardware checks which way segment grows.

② 1 : positive, 0 : negative direction.

→ Segment can be shared b/w address space.

→ Code sharing is still in use in systems today.

→ by extra hardware support.

→ Extra hardware support is need for form of Protection bits.

→ a few more bits per segment to indicate permissions of read, write & execute.

SP PAPER PRODUCT

Segment = (Virtual Address & segMask) >> SEG_SHIFT

Date:

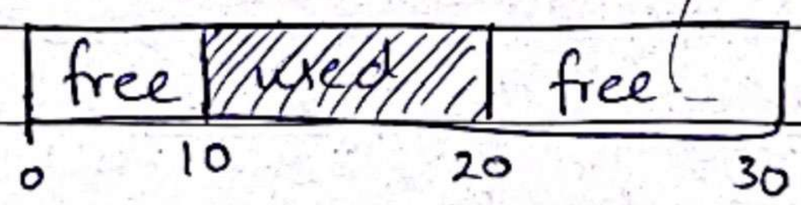
→ Coarse-grained means segmentation in small number.
e.g code, heap, stack.

→ Fine-grained segmentation allows more flexibility for address space in some early system.
→ hardware support with segment table is required.

CHAPTER 17: Free-Space Management.

Splitting (smaller fragments)

→ finding a free chunk of memory that can satisfy the request and splitting it into two.
→ When a request for memory allocation is smaller than the size of free chunks.



free list head → (add:0, len:10) → (add:20, len:10) → NULL.

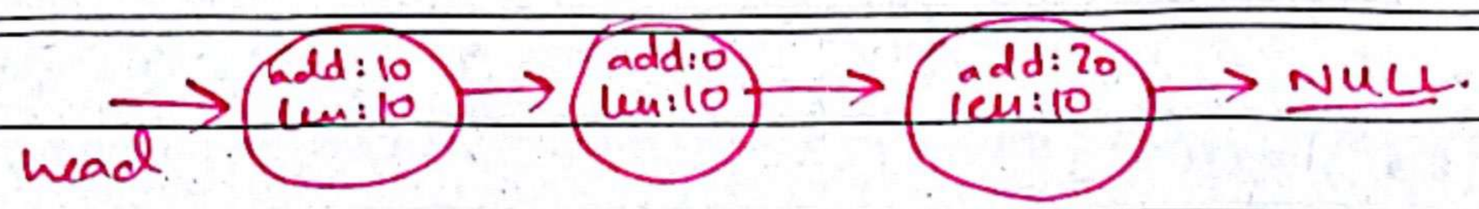
Coalescing

→ if a user requests memory that is bigger than free chunk size, the list will not find such a free chunk.
→ Coalescing: Merge returning free chunk with existing chunks into a large single free chunk if addresses of them are nearby.

Merging two contiguous chunks of memory

SP PAPER PRODUCT

Date:



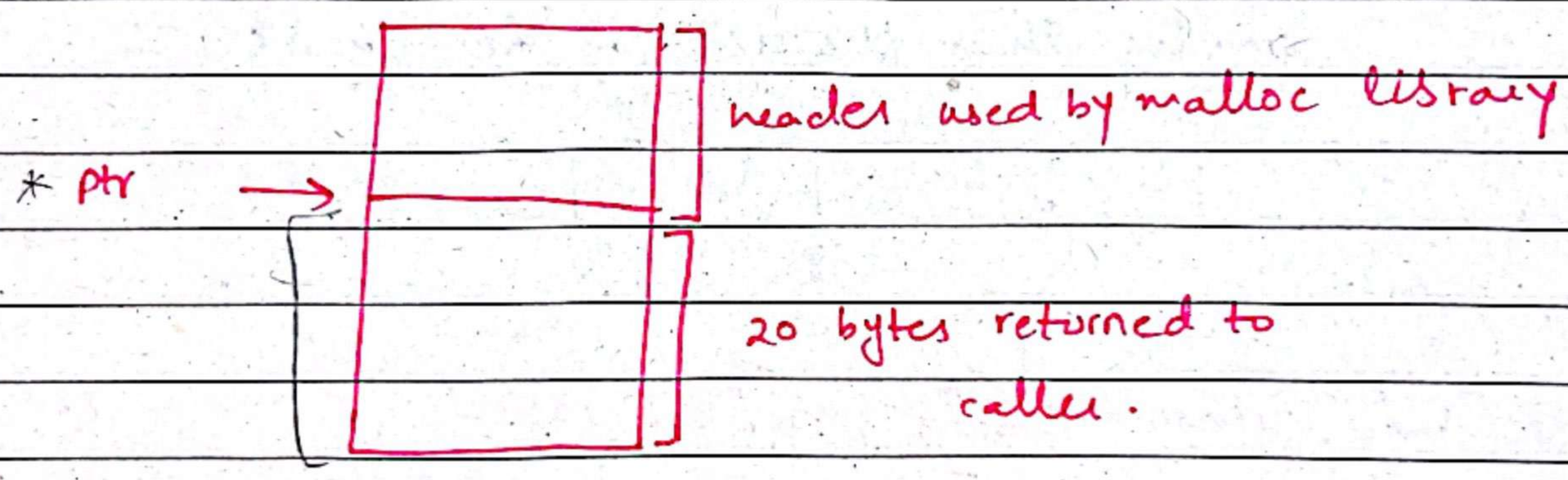
coalescing free chunks.

TRACKING THE SIZE OF ALLOCATED REGIONS.

→ The interface to free(void *ptr) does not take a size parameter.

→ most allocators store extra information in header block.

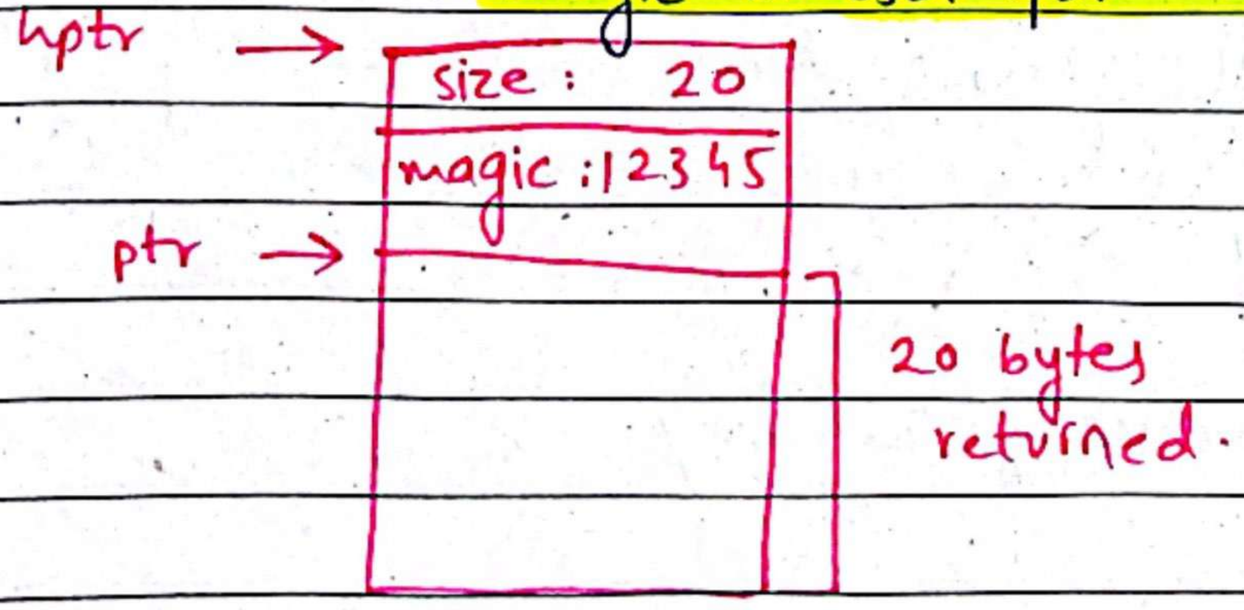
ptr = malloc (20);



Header. contains the size of allocated memory region.

→ additional pointers to speed up deallocation.

→ magic number for integrity checking.



SP PAPER PRODUCT

Date:

→ The size for free region is the size of header plus size of space allocated.

```
void free(void * ptr) {
    header_t *hptr = (void *) ptr - sizeof(header_t);
}
```

→ The memory allocation library initializes the heap and puts the first element of free list in free space.

```
typedef struct __node_t {
    int size;
    struct __node_t *next;
}
```

node of free list.

→ if a chunk of memory is requested, the library will first find a chunk that is large enough to accomodate request.

→ the library will

→ Split the large free chunk into two.

→ one for request & remaining ~~for~~ free chunk.

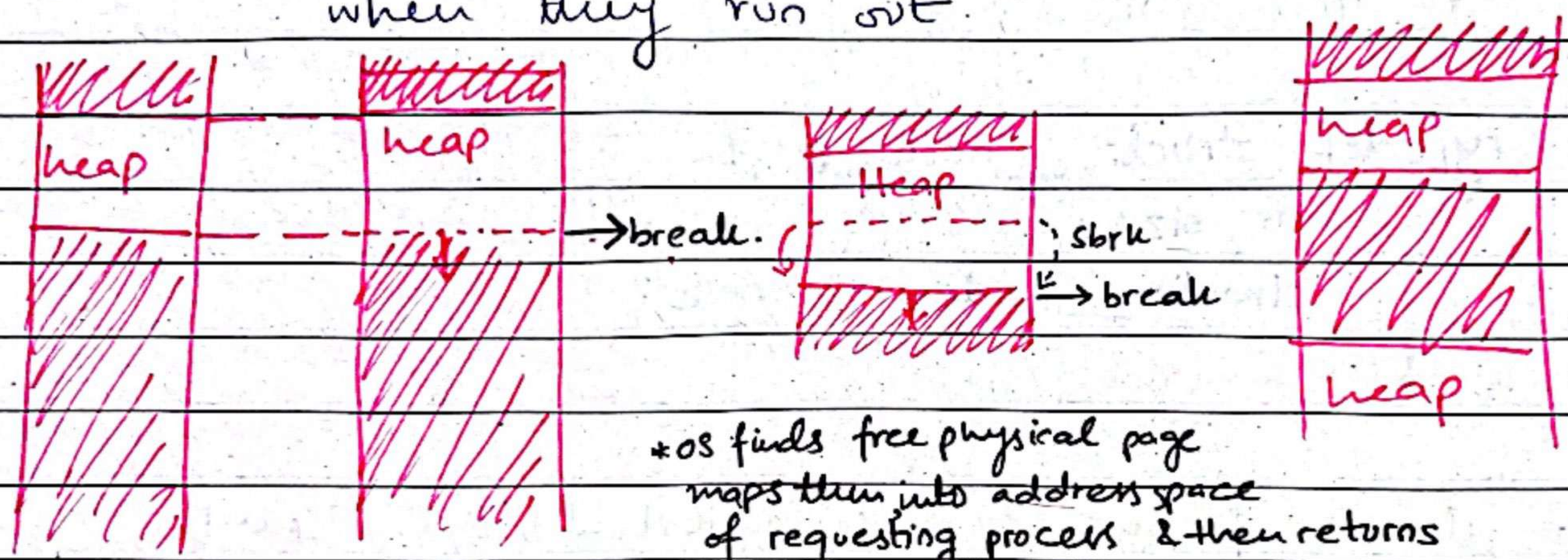
→ Shrink the size of free chunk in list.

SP PAPER PRODUCT

Date:

* Example: a request for 100 bytes by ptr = malloc(100)
  → allocating 108 bytes out of existing one free chunk.
  → shrinking the one free chunk to 3980 (4088 minus 108).

→ Most allocators start with a small-sized heap and then request more memory from OS when they run out.



* OS finds free physical page maps them into address space of requesting process & then returns value of end of new heap.

Managing free space.

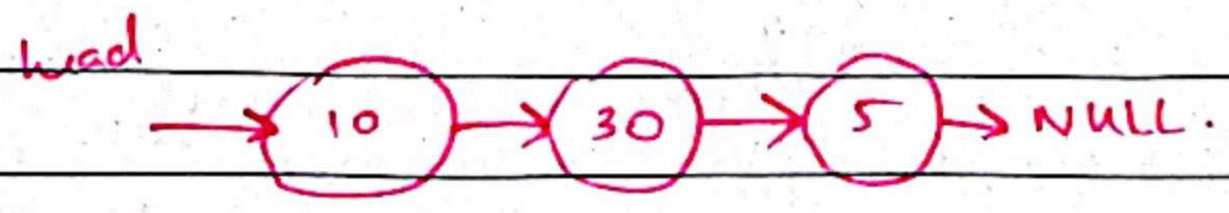
① Best fit

→ finding free chunks that are bigger ~~or~~ than request or big.

→ returning one of $\boxed{\text{smallest in chunks}}$ in the group of candidates.

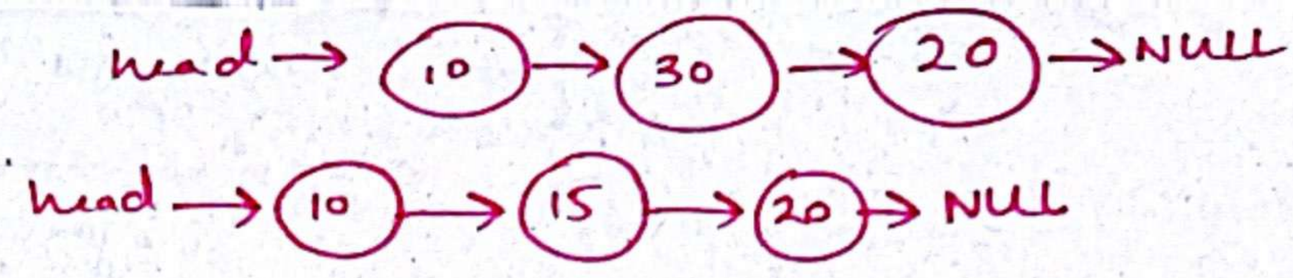
→ scroll through list (takes time)

→ expensive.



SP PAPER PRODUCT

size : 15



Date:

② Worst fit

→ finding largest free chunks and allocation the amount of request.

→ keeping the remaing chunk on free list.

③ First fit

→ finding the first chunk that is big enough for request

→ returning the requested amount & remaing the rest of the chunk.

④ Next fit

→ finding the fist chunk that is big enough for request.

→ Searching at where one was looking at instead of begging of the list.

⑤ Seggregated list. (saves from external fragmentation)

→ keeping free chunks in different size in a separate list for the size of popular request.

→ new complication

→ How much memory should dedicate to pool of memory that serves specialized requests of given size?

→ Slab allocator handles this issue.

SP PAPER PRODUCT

FRAGMENTATION

① Internal.
→ you asked for 15KB but got 16KB one free KB for free space.
(minimized by ~~fixed-size paging~~ segmentation)

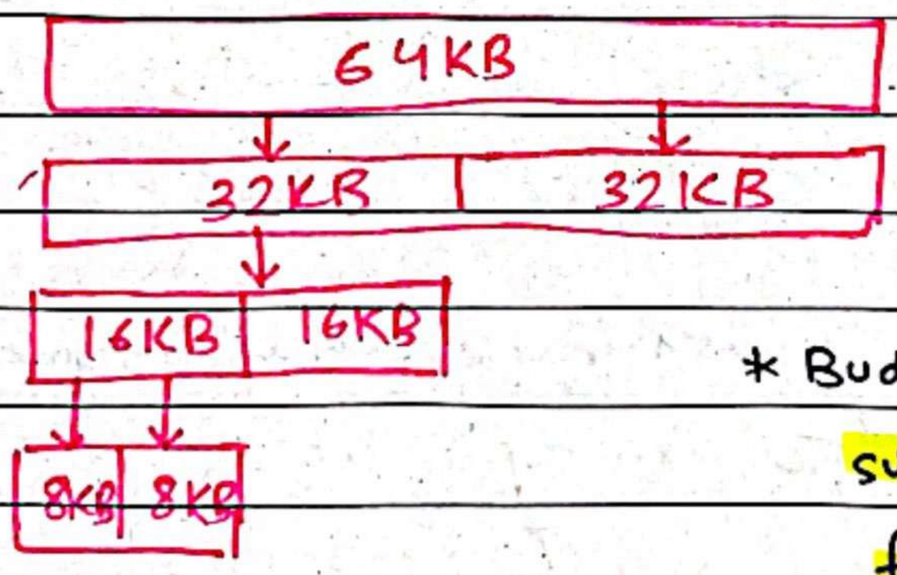
② External.
outside each process (reduced by dynamically allocating pages). (resolved by ~~segmentation~~) paging.

Date:

→ Slab Allocator.

→ Allocate a number of object caches
→ The objects are likely to be requested frequently
e.g locks, file-system inode, etc.
→ Request some memory from more general memory allocator when a given cache is running low on free space.

ⓒ Binary Buddy Allocation.
→ The allocator divides free space by two until a block that is big enough to accomodate the request is found.



* memory chunk will always be a power of 2 size.

* Buddy allocation can suffer from internal fragmentation.
* Buddy system makes coalescing simple.

SP PAPER PRODUCT

Offset = $\log_2$(Total Pages) Total No. of Pages = $\frac{\text{Address Space}}{\text{Page Size}}$.

Page Fram Size = Page Size. Address Space = 32 bit ...

Date:

CHAPTER 18: Paging.

→ Paging splits up [address space] into [fixed-size unit] called a [page].

→ with paging, [physical memory] is also split into same number of pages called [page frame]

→ Paging per process is needed to translate [virtual address] to physical address.

* page table stores page frame.

Advantages:

① Flexibility: Supporting abstraction of address space effectively

② Simplicity: ease of free-space management.

→ page in address space & page frame are same size.

Disadvantages: Additional memory accesses slow down the system

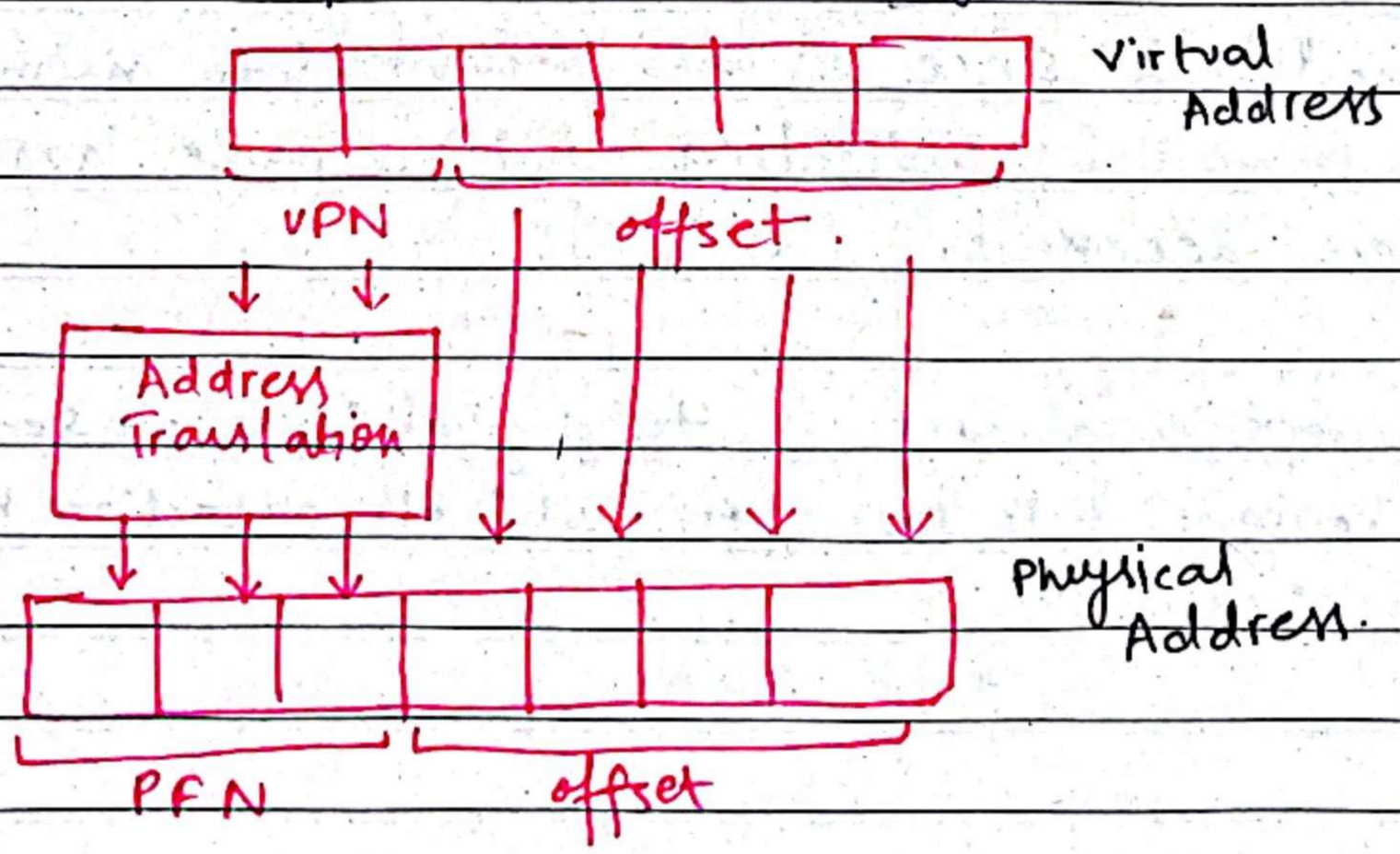
* the size of page is determined by the CPU.

Address Translation

→ Two components in virtual address.

① VPN: Virtual Page Number → PFN: Page frame number.

② offset: offset within page.



SP PAPER PRODUCT

→ every process has it's own page table
→ kept in PCB.
→ every process has it's own PCB.

* Paging splits address to fixed size unit while Page Table maps virtual address to physical address.

Date:

→ Page tables can get awfully large.
→ 32-bit address space with 4-KB pages, 20 bits for VPN.
→ 4MB = 2^{20} entries * 4 Bytes per page table entry
→ Page table for preach process are stored in memory.
→ The page table is just a data structure that is used to map virtual address to physical address.
* an array.
→ The OS indexes the array by VPN & looks up page-table entry. ✓
PFN
valid Bit

→ Valid Bit indicating whether particular translation is valid.
→ Protection Bit indicating whether page could be read from, written to or executed from.
→ Present Bit indicating whether this page is in physical memory or on disk.
→ Dirty Bit indicating whether page has been modified since it was brought into memory.
→ Referenced Bit indicating that a page has been accessed.

* Page table maps virtual page no to physical frame number.
A (PTE) contains info like (PFN), valid bit, protection bit and flags.

SP PAPER PRODUCT

* each entry in Page Table ~~stores~~ represents VPN
→ VPN is the index in Page Table

→ PFN
→ valid bit
* stores key information about specific page in memory.

Page Table Entry.

Date:

→ To find a location of desired PTE, the starting location of the page table is needed.

[→ for every memory reference, paging requires the OS to perform are extra memory reference.] ??

* Advantage of using Page Frame of fixed size in paging.
→ simplify memory management
→ manage memory uniformly
→ maintain simple mapping from virtual to physical.

Q) what happens when memory reference is made & PTE Present Bit = 0 indicates page is not present in physical memory?
→ a page fault occurs, prompting OS to load required page from secondary storage into physical memory.

CHAPTER 19: Paging - Faster Translations (TLBs)

→ How to speed up Address Translation?
→ help comes from hardware
→ Translation lookaside Buffer → hardware → TLB
→ part of chip's Memory Management Unit (MMU)
→ cache used to store virtual to physical address mapping
→ speeds up translation process as it eliminates need to consult page table for each memory reference.

SP PAPER PRODUCT

* CPU would not be doing address translation because alot of clock cycles would be wasted.

Date:

TLB Basic Algorithm

VPN = (virtual Address & VPN_MASK) >> SHIFT.

① extracting VPN from Virtual Address.

(success, TlbEntry) = TLB_Lookup(VPN).

② Check if TLB holds translation for this VPN.

→ Hit: TLB holds the translation, extract ~~VPN~~ PFN from relevant TLB entry, concatenate that onto the offset and access memory (Physical Memory)

→ Miss: does not find translation access page table to fetch translation, update TLB and retry

Phy

Accessing an Array.

→ demonstrates the impact of TLB misses and hits when accessing array elements.

→ Spatial locality: Accesses to elements within same page benefit from TLB hit.

as elements are tightly packed

→ Temporal locality: leads to better hit rates upon repeated accesses.

→ larger page size reduces the number of TLB misses.

SP PAPER PRODUCT

Date:

HANDLING TLB MISSES — ① hardware ② OS.

CISC

→ The hardware would handle the TLB miss entirely.

→ to do this hardware has to know exactly where the page tables are located in memory via Page-Table Base Register, as well as their exact format on a miss

→ reads PFN, updates TLB and retries instruction.

RISC

→ Software-Managed TLB

→ on TLB misses, the hardware raises exception.

→ pauses current instruction stream, raises priviledge mode to kernel mode and jumps to trap handler.

→ reads PFN, use special "priviledged" instruction to update TLB and return from trap.

→ return_from_trap (exception)

→ different from return_from_trap systemcall as it returns to the instruction which raised exception, executing same instruction again.

TLB Entry. VPN | PFN | other bits.

→ Typically 32, 64 or 128 entries. and be what is called fully associative

→ search can be performed anywhere in TLB in parallel.

SP PAPER PRODUCT

Q) What if the instruction that accesses a page table for PFN results in TLB miss?

→ CPU performs a ~~walk~~ page table walk, it traverses page table hierarchy to find PFN in memory.

• CPU will go step by step in Page Table to find physical address, once found, its stores result in TLB.

Date:

→ Other Bits:

① Valid
② Protection
③ Dirty
④ Address Space Identifier

TLB Issue : Context Switches.

→ TLB contains virtual to physical translations that are only valid for currently running process; these translations are not meaningful for other processes.

→ As a result when switching from one process to another, the hardware or OS (or both) must be careful to ensure that the about-to-be-run process does not accidentally use translations from some previously run process.

Solution :

① Flush: Clear TLB entries during context switches.

② Tagged TLBs : Add a Address Space Identifier (ASID) → similar to PID().

→ needs hardware support. ASID of current process has to be updated on context switch.

) Replacement Policy

→ When we are installing a new entry in the TLB, we have to replace an old one.

SP PAPER PRODUCT

Date:

→ But which are to replace?

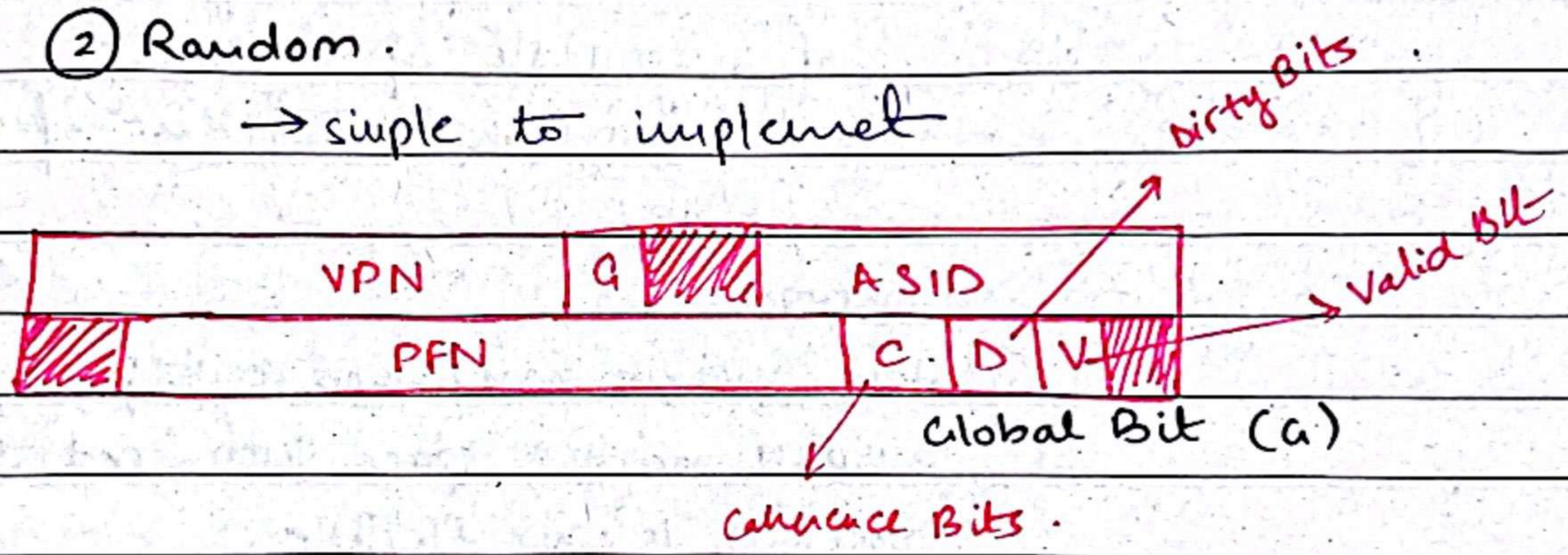
① LRU.

→ loop over n+1 pages in a TLB of size n, each loop we have a miss

→ example: we have an array of 100 elements, TLB with 36 bytes per page, each loop accesses 10 array elements.

② Random.

→ simple to implement



CHAPTER 20: Paging: Smaller Tables.

→ Page Tables are too big and thus consume much memory

Solution:

① → Use larger pages to reduce the size of page tables and hence storage requirement

LARGE PAGE SIZE → REDUCED SIZE OF PAGE TABLE → Drawbacks:

↓

reduces TLB misses

① Internal fragmentation: (Applications may not fully utilise large pages, wasting memory).

SP PAPER PRODUCT

No of Segments ⇒ that many base/bounds pair.

Level 1 Page Directory
Level 2 Page Table
Level 3 Page Frame

Date:

(2) → Hybrid Approach.

→ Combine paging with segmentation for efficiency. to only store entries with data.

Virtual Address
↓
Page Directory Index
→ points to Page Directory
↓
Page Table Index
→ Points to Page Table
↓
offset → points to physical page.

→ Advantages:

(1) Reduces unused page table entries.

→ have one page table per logical segment

→ base register : contains physical address of page table of segment

→ bounds register: indicates end of page table

→ Disadvantages:

(1) Have multiple base/bound registers.

(2) dividing address space into fixed size portions is not flexible.

(3) Variable sized page tables depending on segment size.

(3) → Multi-level Pages.

→ Splits a large page table into multiple levels to save space. (page sized fragments)

→ Turns page table to something like a tree.

(1) Chop up page table to page sized fragments.

(2) if an entire page of page table entries is invalid don't allocate that page

(3) to track whether a page of page table is valid use new structure called PAGE DIRECTORY

→ tells where page of page table is

→ which pages are valid.

SP PAPER PRODUCT

Top level ① virtual address to base address
② ~~Maps portion of virtual address to physical frames.~~

2nd level: Page Number to Frame Number properly)

Date:

→ Advantages
① Page table size are reduced.
② Page tables can be distributed in non contiguous pages.

→ Disadvantages.
① Complexity of implementation.
② On TLB miss, we've to access RAM 3 times.
→ can lead to larger translation times.

Example: Crux.
→ instead of allocating 16 pages of linear page table we allocate 3 pages. ① Page directory ② for page table chunks.

INVERTED PAGE TABLE. (maps physical frames to virtual pages)
→ One page table for whole system. Table size (Small).
→ indices are PFN's instead of VPNS. Lower overhead.
→ PTE contains: PID, VPN
→ have to do linear search.
→ hash tables may help reduce search time.
→ requires less memory since table shared across all processes

CHAPTER 21: Swapping Mechanisms.
→ The address space portions that are currently not in use/great demand are to be placed somewhere.
→ served by hard disk
→ usually we assume that process address space can fit within available physical memory
→ Pages are moved to hard disk if they can't be accomodated in system memory.
→ Slow system response if there is a lot of page swapping.

SP PAPER PRODUCT

designated area on hard disk used by OS as an extension of physical RAM when it becomes full.
① when OS runs out of memory it moves less frequently used memory pages to swap space.
② when the required pages are needed again, they are swapped back into RAM.

Date:

SWAP SPACE. (Temporary Storage).

→ reserve some space on the disk for moving pages back & forth.

→ OS needs to remember to the swap space, in page-sized unit.

Present Bit

→ Add some machinery higher ~~end~~ up in the system in order to support swapping pages to and from disk.

1 → page present in physical memory

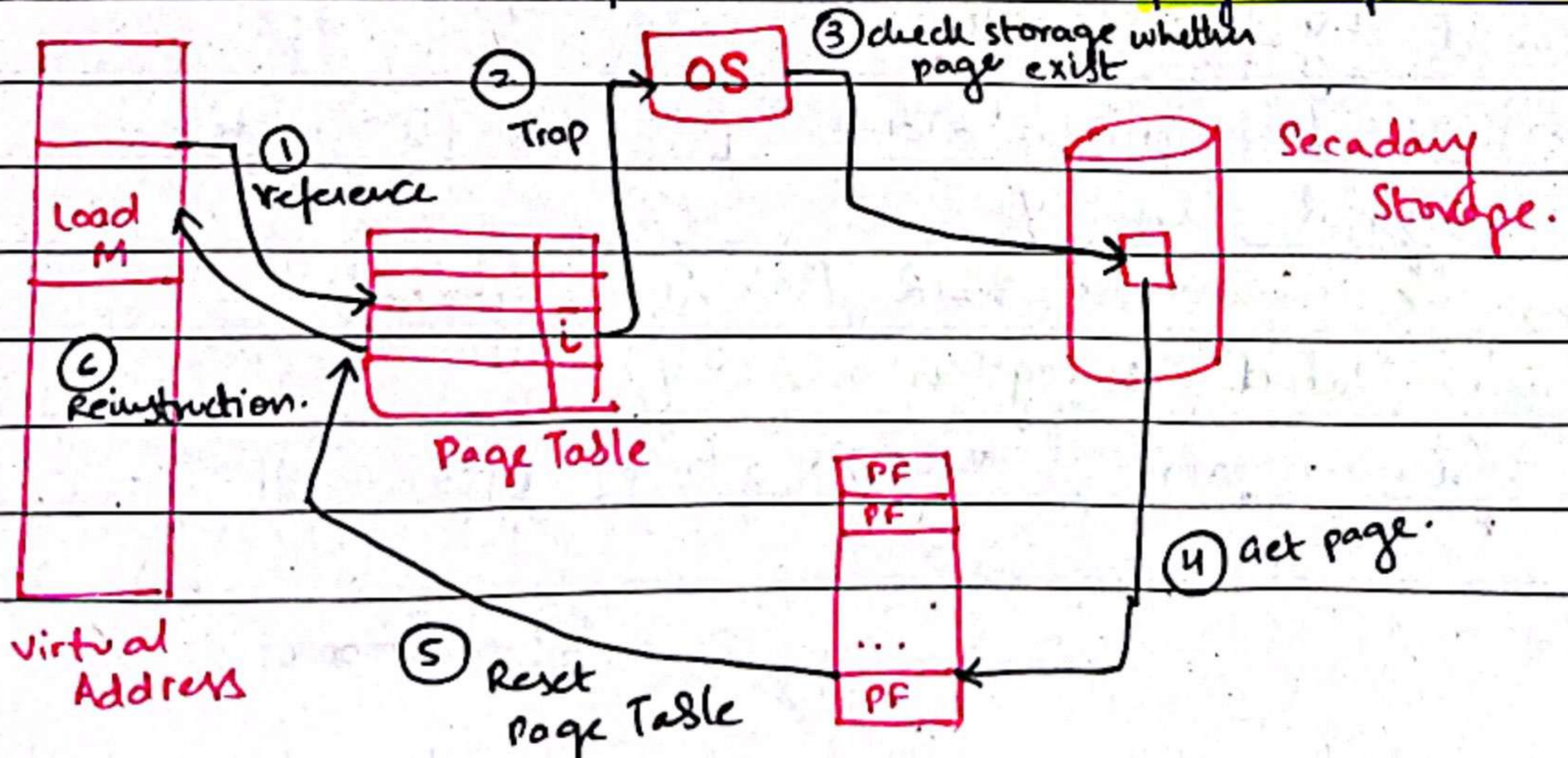
0 → page not in memory but rather on disk.

Page Fault :- Occurs when a program tries to access page not present in physical memory

What if Memory is full?

→ OS might like to first page out one or more pages to make room for new pages the OS is about to bring in.

→ The process of picking a page to kick out, or replace is known as page replacement policy.



SP PAPER PRODUCT

PAGE REPLACEMENT POLICY.

1. least Recently Used (LRU)
2. First in First out (FIFO).
3. Optimal Page Replacmet
4. Least Frequently Used (LFU)

HIGH WATERMARK (HW)
→ represents maximum allowable memory usage before OS takes action.
→ if it crosses theshold OS triggers aggressive actions (swap out, kill process needed)

LOW WATERMARK (LW).
→ represents minimum memory level the OS tries to maintain
→ drops below this theshold, OS starts reclaiming memory in aggresive way

Date:

When replacements really occur ?

→ High Watermark (HW) and Low Watermark (LW) to help decide when to start evicting pages from memory.

→ Swap Daemon, Page Daemon.

→ There are fewer than LW pages available, a background thread that is responsible for freeing memory runs.

→ The thread evicts pages until there are HW pages available.

CHAPTER 22 : Swapping Policies.

→ Memory pressure forces OS to start paging out pages to make room for actively-used pages.

→ Deciding which page to evict is encapsulated within replacement policy of the OS.

Cache Management.

→ goal in picking a replacement policy for this cache is to minimize number of cache misses

→ the number of cache hits and misses let us calculate average memory access time (AMAT)

$$AMAT = (P_{Hit} * T_M) + (P_{miss} * T_D).$$

P_{Hit}: probability of finding data in cache

T_M: cost of accessing memory

P_{miss}: probability of not finding data in cache.

T_D: cost of accessing disk.

SP PAPER PRODUCT

best policy leads to fewest number of misses

imaginative policy.

order doesn't matter

Date:

① The Optimal Replacement Policy.

→ leads to fewest number of misses overall.

→ replaces page that will be accessed furthest in future. → something not used in future

→ resulting in fewest-possible cache miss

→ serve only as a comparison point to know how to close we are to perfect.

Access	Hit/Miss?	Result.
0	Miss	0
1	Miss	0,1
2	Miss	0,1,2
0	Hit	0,1,2
1	Hit	0,1,2
3	Miss	0,1,3
0	Hit	0,1,3
3	Hit	0,1,3
1	Hit	0,1,3
2	Miss	0,1,2
1	Hit	0,1,2

$$\frac{\text{Hits}}{\text{Hits} + \text{Miss}} = 54.6\%$$

② Simple Policy: FIFO.

→ Pages were placed in queue when they enter system.

→ when a replacement occurs, the page on tail of queue (the "First-in" pages) is evicted.

→ Simple to implement, but can't determine importance of blocks.

Access	Hit/Miss	Replacement
0	Miss	0
1	Miss	0,1
2	Miss	0,1,2
0	Hit	0,1,2
1	Hit	0,1,2
3	Miss	1,2,3 0
0	Miss	2,3,0 1
3	Hit	2,3,0
1	Miss	3,0,1
2	Miss	0,1,2
1	Hit	0,1,2

SP PAPER PRODUCT

*Random is wast

Date:

Belady Anomaly: We would expect the cache hit rate to increase when the cache gets larger. But in this case, with FIFO it gets worse. hit rate decreases

③ Random.

→ Picks a random page to replace under memory pressure

→ it doesn't really try to be too intelligent in picking which blocks to evict.

→ Random does depends entirely upon how lucky Random gets in it's choice.

Access	?	Result.
0	Miss	0
1	Miss	0,1
2	Miss	0,1,2
0	Hit	0,1,2
1	Hit	0,1,2
3	Miss	1,2,3
0	Miss	2,3,0
3	Hit	2,3,0
1	Miss	2,0,1
2	Hit	2,0,1
1	Hit	2,0,1

* Sometimes, Random is as good as optimal achieving 6 hits an example trace.

worst, not better than FIFO, one or two case optimal.

Using History

Recency → The more recently a page has been accessed, the more likely it will be accessed again. LRU

Frequency → If a page has been accessed many times. It should not be replaced as it clearly has some value. LFU

SP PAPER PRODUCT