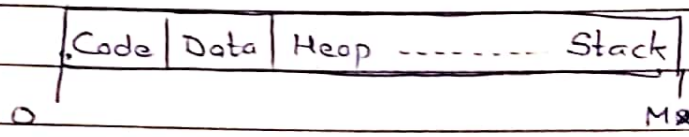
24/1

Memory Management

- Memory allocated to a process is usually not contiguous, for efficiency
- Every process believes it has memory $\in [0, \text{some max value}]$
  - 'Virtual Address Space'



- Virtual address = What programmer perceives
  eg - What &x will ↳ assume.
  ∴ CPU also deals with virtual addresses only

- OS must maintain a map: Virtual addresses → Real addresses
  - Map is stored on PCB.

→ Memory Management Unit. (MMU)
- Performs address translation. Present between CPU and RAM.
- Input :- Virtual address from CPU
  Output :- Real address to RAM
- OS manages, modifies MMU

CPU - Cache - MMU - RAM

→ Size of virtual address space
- Dictated by number of bits in PC

eg 32-bit architecture has 32-bit PC. ⇒ 2^{32} bytes of virtual memory (4GB)

→ Granularity of chunks that virtual memory is divided into, each of which gets a contiguous physical address space.

1 Segmentation. ↳ Code, data, etc

- Divides virtual memory into segments, each of which is mapped to contiguously to physical memory

Global variables :- Data ~~(about)~~ section
Local variables of a function - Stack
Dynamic variables - Heap

Page No.
Date

- Maintains a table, which is given to MMU

	Size	Virtual base (start)	Physical base (start)
Code segment	50	100	3000
Data segment	50	150	6000

Size is fixed

- ∴ Sizes of code segment, data segment, etc are different for different processes and may not be equal to allocated size.

Better 2 Pages

- Divides virtual memory into 'logical pages' of equal size, without caring if it is code, data, etc. and maps them to 'physical frames' of same size in physical memory.
- Granularity of allocation is 'page'
  - Some wastage of space, as each process might not use the entire page
    'Internal fragmentation' IDGAF
  - * Wastage caused by variable segment sizes in segmentation was the OS's headache
    'External fragmentation' ∴
- 'Page Table' : Logical Page No. → Physical Frame No.
  - Stored in PCB, given to MMU.
- Typical page size = 4kB

→ Expansion of a process's allocated memory

- During fork, child is allotted same memory as parent.

- Initially, there is empty memory between heap and stack.
  - Stack is automatically built up by OS.
  - System call must be made to increase heap size.
    'brk() or sbrk()'

PTO:

Page No.
Date

Last address value

* 'Program break' $\triangleq$ Code + Data + Heap = in heap.
  - brk() or sbrk() expands program break (expands heap)
- Expansion of heap is in granularity of one page.
- There actually is large separation between heap and stack.

* mmap() :- System call which returns address of any one empty page from anywhere in between (between heap and stack).

1/2 → Kernel Code Storage

= 1 GB out of 4 GB RAM virtual address space of every process
- Stored at the beginning of RAM.
- Even though you know physical addresses of kernel code, MMU can only take virtual address as input.
- Kernel code must be entered into page table instead
- Addresses 3GB to 4GB in virtual address space of every process contain entire Kernel code, including PCBs of 'all' processes, etc.
- These addresses in every process are accessible only when in kernel mode

* It is impossible to access memory via actual physical address
- Have to allot pages and enter into page table.

* C library is in the physical RAM. Entries are made into all processes' page table. (in 'code' section)
- C library is a 'shared executable'

→ Obtaining shared memory for IPC

- System Call :- ~~char~~ char * a = ~~shm~~ shmget();
- Granularity of 'page's.

→ MMU

1. Translates virtual addresses (requested by CPU) to physical memory
2. Checks permissions :-

eg - Code memory of process must be Read-Only

eg - Kernel code must be inaccessible in user mode

eg - Reduces segmentation fault if an unallocated virtual address is requested.

eg - Virtual address beyond Virtual Space is requested :- Error

- Raises a TRAP instruction if any permission is violated, and then jumps to kernel code before returning from trap.

→ OS

1. Maintains a 'free list' - containing all physical addresses that ~~do not~~ aren't used.
2. Allocation - via system calls : fork(), exec(), etc.
3. Constructing Page Table for PCB of every process, and updating ~~&~~ after context switch
4. Setting Page Table for MMU
   - Give MMU new Page Table after every context switch
     → (starting address of PT)
     → physical, not virtual

* Copy-on-Write Fork

- Don't copy immediately. Make different copy ~~for~~ for child only when one of the child and parent wants to write back some value (change some variable)
- Reading, printing etc can be done on same memory image.
- Beneficial because most children want to exec() any way.

- Both processes still have different CPU context from the beginning. But child will still point to parent's physical memory

- Implementation :-

Data of forked process is made Read-Only.
When parent/child tries to write, MMU will raise trap.
Then memory image is copied

eg 8-bit Virtual Address Space :: Each process has virtual addresses (0, 255)

Page Size = 16 ⇒ 16 logical pages.

eg - Accessing virtual address 35 ≡ 0010 | 0011

0010 → Page No. (2)
0011 → offset (3)

- 2^k page size :: k-bit offset

6/2

- If virtual address space has 2^v bytes.
  page table has 2^{v-k} logical page entries.

- i^{th} entry in page table contains :-

1) Physical frame number (PFN) - which ~~page~~ frame is allocated to i^{th} logical page.
2) Permissions - RO, R/W ....
3) Valid - if i^{th} page is being used
4) Present 5) Dirty 6) Accessed

☆ eg Page size = 4kB

Every entry in PT takes up 4 bytes

No of pages = $\frac{2^{32}}{4kB}$

∴ PT size = $\frac{4 \text{ bytes} \times 2^{32}}{4B} = 4MB$

4B

→ Outer Page Table

- Page Table is divided into chunks, because it is too big (4MB)
- Outer page table = Mapping from Chunk # → Frame #
- So now, for obtaining physical address, get frame from OPT which will give you frame # of PT, refer that to get actual physical address
  → where chunk of PT is in physical memory
- 32-bit systems work with 2 level PT's, 64-bit requires (7) levels.
  6

Eg 32 bit Virtual address space, 4 kB page size, 4B page table entry size

Page Table size = 4MB

∴ Page Table itself needs to be stored in physical memory as 4 kB chunks

- No. of chunks = $\frac{4MB}{4kB} = 2^{10}$
- These chunks are stored in outer page tables as 2^{10} frames
- Each OPT entry is 4B

∴ OPT size = $2^{10} \times 4B$
= 4 kB

∴ 4 kB OPT can be contiguously stored in physical memory, so we don't need other levels.

- The MMU is given the address to outermost page table.
- Address to be fetched

10	10	12

  - First 10 bits will tell which chunk of innermost page table to look at.
  - Next 10 bits tell ~~the Base of physical address~~ look the entry in innermost page table, that will give base physical address of page

P.T.O.

- TLB is managed entirely by hardware (not OS)
- The cost of TLB misses is more if no. of levels in paging system is more.

Page No.
Date

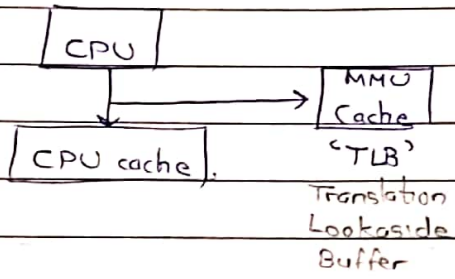
Last 12 bits give offset within that page.

* 64-bit address :-

2	10	10	10	10	10	12

- Looking up into multiple tables takes more memory accesses (∵ PT's are stored in memory)
∴ Most modern MMUs use cache.

→ MMU Cache :- (TLB)



MMU cache stores most recently used page table entries (and not the actual data)

- MMU is requested only if CPU cache misses.
- MMU cache will give the stored mapping so that MMU can reach final physical address asap.

- Every time a process context switch happens, page table is switched and entire MMU cache needs to be flushed out
  - Some caches could store entries for more than one process (and identify mappings based on PID)

- TLB uses LRU for replacement.

* Higher Page Size :-
  - Smaller Page Tables ☺
  - Higher TLB cache hit rate ☺
  - Higher internal fragmentation ☹ (wasted space within a page)

8/2

→ **Swap Space**

* Demand Paging – OS allots new pages to processes if they are unused.

- If computer has no unused memory left, some pages of some processes are stored in 'swap space' of ROM (disk) to clear up space in RAM.
  - Current running process cannot be in swap space

- Every entry in Page Table contains PFN, permissions, valid bit (seen before).
  It also has a 'present' bit – '1' if that page is in memory
  '0' swap space
  - If present == 0, CPU cannot request for that address.
  - Present bit is meaningless if valid = 0.

→ Most OS's do not immediately allocate physical frames to data pages of a process, even though a page table entry with valid = 1, present = 0 is created.
Actual physical frame is allocated (present → 1) only when address is requested for the first time.
eg – You expand heap and put a variable there (valid → 1, present = 0)
Present → 1 only when that variable is accessed.

  - When MMU tries to translate a valid but not present address, it raises a trap called 'Page Fault'
    - OS must bring the page to memory, update Page Table. P.T.O
    - Process is blocked in the mean while. Becomes Ready when page is brought to memory. Continues execution.

→ Memory Accesses

1. CPU requests for a virtual address
2. It checks in data cache.
3. If not in cache, it tries to get it from memory.
   Access MMU.
   Check if TLB can give you frame number. If yes, fetch from memory (single access)
4. If not in TLB, traverse Page Table(s) to get physical frame (requires multiple memory accesses)
5. If entry in PT is not present, raise a Page Fault.
   Then repeat.

Both data cache and TLB have ≥ 90% hit rate.

- For addressing a Page Fault, OS must bring requested page to memory
  If memory is full, a 'Victim Page' must be 'evicted' to swap space to make room for requested page. (Total 2 memory accesses)
  - Decision of victim : 'Page Replacement Policy'
    1) Random
    2) ~~LRU~~ FIFO - Could be bad because you might evict a frequently used page.
    3) LRU - Good
    4) Approximate LRU :-
       Every Page Table Entry also has an 'Accessed' bit (1 if address in that page was accessed).
       Evict the page that is not accessed.
       - Accessed bit is periodically (~ ms) reset to 0 by OS
       - OS can memorize this periodic History of Accessed bit to decide which page to evict
  - Some OS's will proactively evict a few victim pages before a Page Fault occurs (saves one memory access)

Page No.
Date

** Page Table entries also have a 'dirty' bit
(1, if page has been modified and must be written back to memory)
eg - dirty = 0 for code pages that are never modified.
- If dirty == 0, page is simply erased while evicting
Dirty bit could influence victim decision.

- Some pages are not evicted :- 'Pinned pages'
eg - Kernel code that handles Page Faults
eg - Pages that have very recently been fetched from disk.

* ps command in shell shows how much virtual and physical memory a process uses

13/2

→ Memory Allocation and Freeing

• 'Bitmap' - A table maintained by OS, one entry for each page.
Entry = 1 if that page is busy, = 0 if not.
- Hassle to scan through and maintain.

Better • 'Linked List' = 'Free list' :
- Every free page contains PFN of next free page.
- Begin traversing from the 'head'
- The free list need not be in sorted order of PFN
- When you allocate free page, remove it from free list and move head to its next free page
- Newly freed pages are added to the end of free list.

→ Variable Size Memory Allocation.

- malloc allots requested number of bytes to expand heap within a page.

- Variable size of chunks for allocation - 'Splitting' and 'Coalescing'

eg Initially, entire 4kB of page is free, with 'head' pointing to start of page.

```
int * x = malloc (20); // Allocate 20 bytes
                       // 'x' the pointer to the start of the
                       // allocated memory.
free (x);              // frees up 20 bytes. Adds to free list
                       // If not freed within that block, x will
                       // be deleted ⇒ Memory Leak.
```

(within same block)

- While freeing up, free() needs to know how many bits were allotted in the first place.
  - For every allocation, malloc() stores a 'header' of h bytes to store the size of the chunk being allocated. (same for one implementation of malloc)
  - The header also contains a 'magic' bits : Random number given to that chunk before allocation.
    - Can detect unintentional errors, like changing the value of 'x' or changed the size, and tried to access memory out of this chunk.

Page No.
Date

- Coalescing -

Periodically, consecutive small chunks in the page which are free need to be merged into a big free chunk.

Otherwise we wouldn't be able to serve a request of large no of bytes.

- If there are multiple free chunks of size ≥ requested size, which one of those should be allotted?

1) Best fit - Choose the chunk closest in size to requested size.
2) First fit - Choose the chunk that is first in free list.
3) Worst fit - Choose the largest chunk

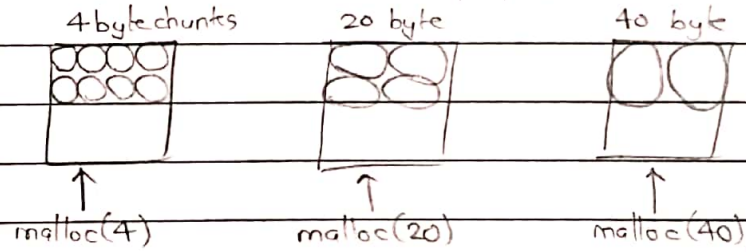
(logic - what remains is still of respectable size)

- Actual implementation = combination of three fits

- "Slab Allocator" : Different implementation, - fixed sizes

Heap is made of several pages,

In this implementation, each heap page is divided into chunks of same size (size is same for a page)

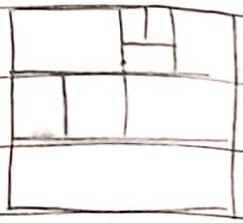


- Much faster than normal malloc
- Within a page, this becomes fixed size memory allocation as before

* The kernel uses slab allocation. ~~One~~ Pages of PCB-sized chunks are used to allocate chunks for newer PCBs.

∴ Cannot allocate any other size for chunks.

Page No.
Date



- Buddy Allocator
  - More flexible than Slab Allocator.
  - Chunk sizes must be in powers of 2 bytes
  - Easier splitting and coalescing.

  ∴ If you want 33 bytes, you need to allocate 64 byte chunk

- Comparison

  Flexibility (∵ Performance) : General > Buddy > Slab
  variable size (user) > 2^k size (kernel) > fixed size (kernel)