CS 347M (Operating Systems Minor)

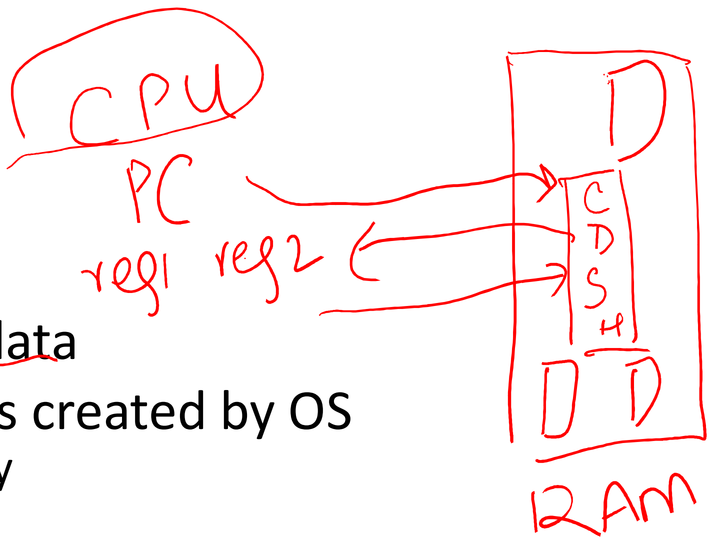
Spring 2022

Lecture 7: Introduction to virtual memory

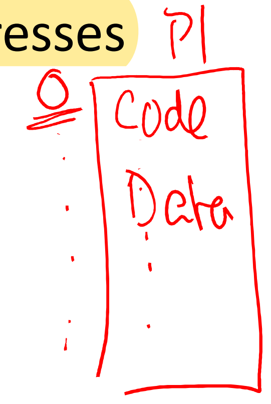
Mythili Vutukuru

CSE, IIT Bombay

Memory management in OS



- User program executable = code + compile-time data
- When program is run, memory image of process is created by OS
  - Code + data from executable loaded into main memory
  - Extra memory allocated for stack and heap
- CPU begins executing process, fetches code/data using their addresses
- What are these memory addresses? Who allocates them?
- Code+data in memory image of every process is assigned virtual addresses starting from 0
- CPU fetches code/data using virtual addresses
- OS knows physical addresses (actual memory locations of code+data)
- OS manages memory, helps to translate virtual to physical addresses



Why virtual addresses?

- Because real view of memory is messy!
- Earlier, main memory had only code of one running process (and OS code)
- Now, multiple active processes timeshare CPU
  - Memory images of many processes must be in memory
  - Memory allocation can be non-contiguous
- Need to hide this complexity from user
- Also, physical address not known at compile time

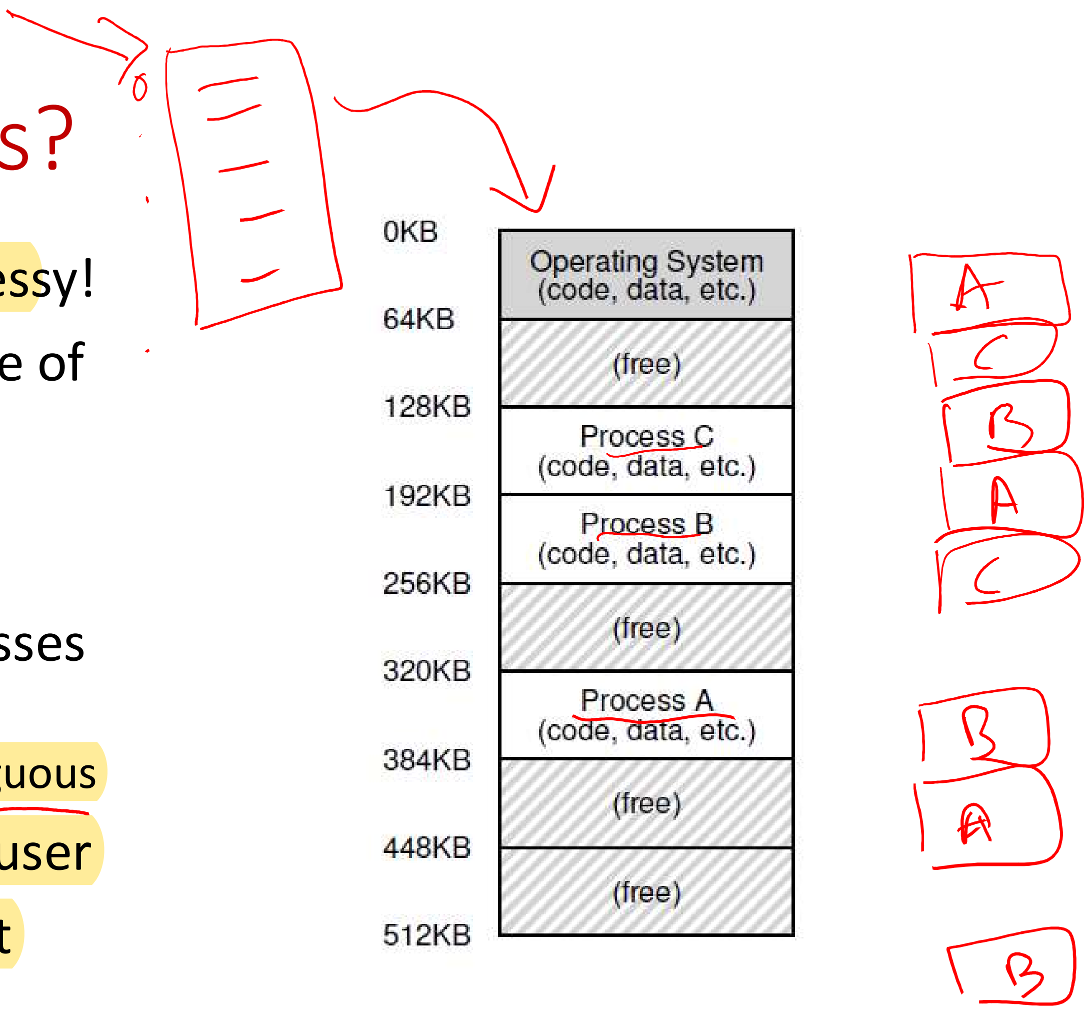

0KB
64KB
128KB
192KB
256KB
320KB
384KB
448KB
512KB
Operating System (code, data, etc.)
(free)
Process C (code, data, etc.)
Process B (code, data, etc.)
(free)
Process A (code, data, etc.)
(free)
(free)
A
C
B
A
C
B
A
B

Figure 13.2: Three Processes: Sharing Memory

Abstraction: (Virtual) Address Space

- Virtual address space: every process assumes it has access to a large space of memory from address 0 to a MAX
- Contains program code (and static data), heap (dynamic allocations), and stack (used during function calls)
  - Stack and heap grow during runtime
- CPU issues loads and stores to virtual addresses

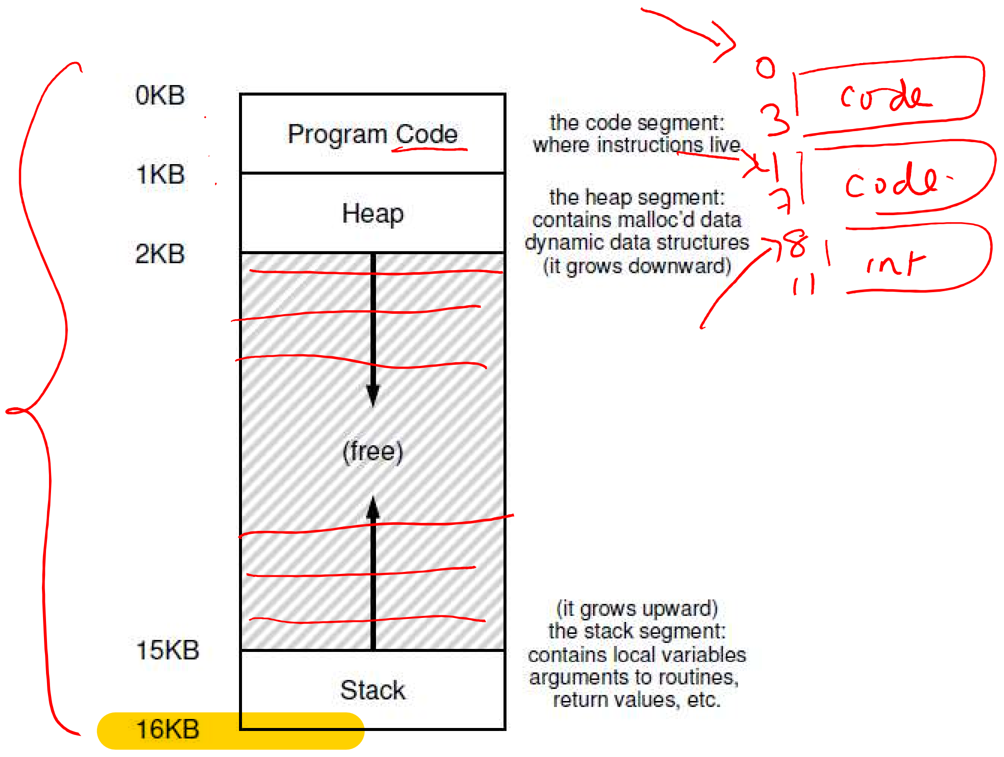
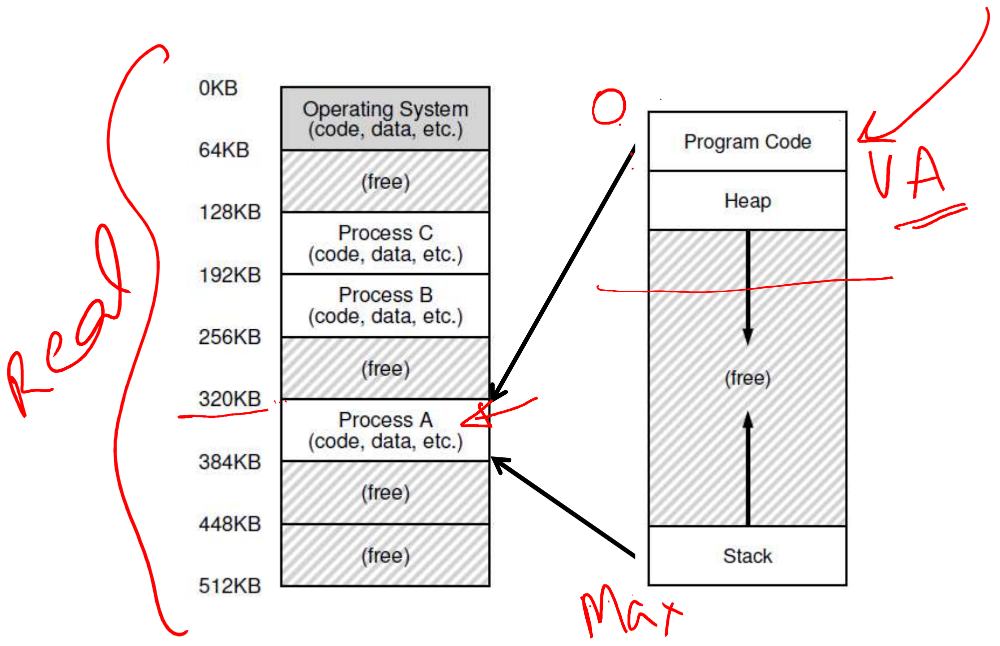

0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack
the code segment: where instructions live
the heap segment: contains malloc'd data dynamic data structures (it grows downward)
(it grows upward) the stack segment: contains local variables arguments to routines, return values, etc.
0
3
code
4
7
code
8
11
int

Figure 13.3: **An Example Address Space**

How is actual memory reached?

- Address translation from virtual addresses (VA) to physical addresses (PA)
  - CPU issues loads/stores to VA but memory hardware accesses PA
- OS allocates memory and tracks location of processes
- Translation done by memory hardware called Memory Management Unit (MMU)
  - OS makes the necessary information available

VA→PA



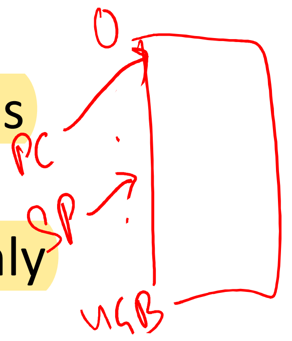
Real
0KB
64KB
128KB
192KB
256KB
320KB
384KB
448KB
512KB
Operating System (code, data, etc.)
(free)
Process C (code, data, etc.)
Process B (code, data, etc.)
(free)
Process A (code, data, etc.)
(free)
(free)
0
Program Code
Heap
(free)
Stack
Max
VA

Virtual and physical address spaces

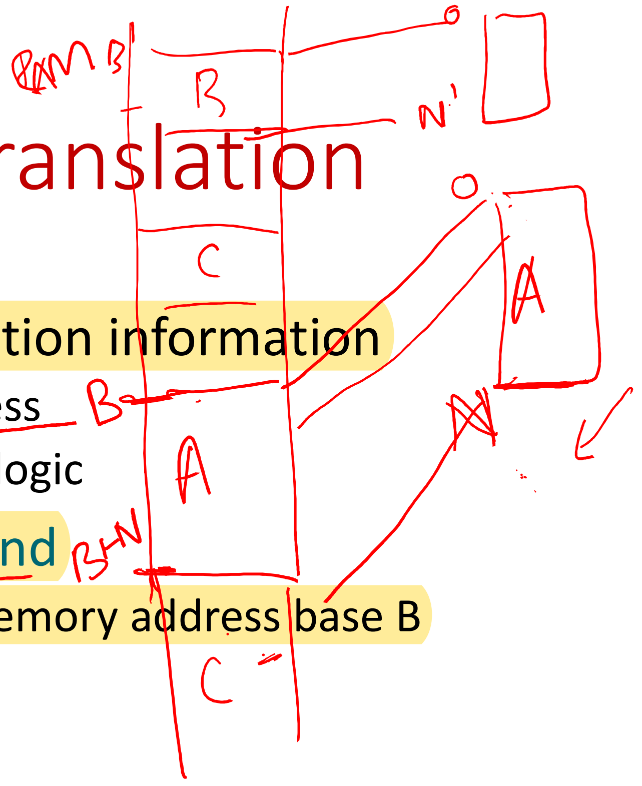
$2^{10} = 1K$
$2^{20} = 1M$
$2^{30} = 1G$

- Virtual address space: set of virtual addresses available to a process 2^{32} bytes
  - 0 to a max value (2^32 = 4GB in 32-bit OS) PC = 32 bits
  - Process can access code/data in its virtual address space
- Physical address space of a system: physical memory addresses in RAM

- Why do we need virtual addresses?
  - Physical address of code/data not known at compile time ✓
  - Memory allocated to a process can be non-contiguous, hide this detail from user ✓
  - Can control which memory a process can "see", useful for isolation
- PC, SP
- Addresses in CPU registers and pointer variables = virtual addresses
  - User only sees virtual addresses, not physical addresses
- But memory hardware can access data using physical addresses only



Memory allocation and address translation



- OS allocates physical memory to a process, has translation information
  - It knows which virtual address maps to which physical address
  - Memory allocation method determines address translation logic
- Simplest form of memory management: base and bound
  - Place entire memory image [0,N] contiguously starting at memory address base B
  - Virtual address X translated to physical address B+X
  - Access to virtual addresses beyond N will not be permitted
- Who does address translation? A piece of hardware called Memory Management Unit (MMU)
  - Every memory access by CPU is translated by MMU before reaching RAM
  - OS provides information to MMU (e.g., base and bound) for translation/error checking

B N

$$\begin{tikzcd} CPU \arrow[r, "X"] & MMU \arrow[d, "B+X"] \\ & RAM \end{tikzcd}$$

A simple example

- Consider a simple C function

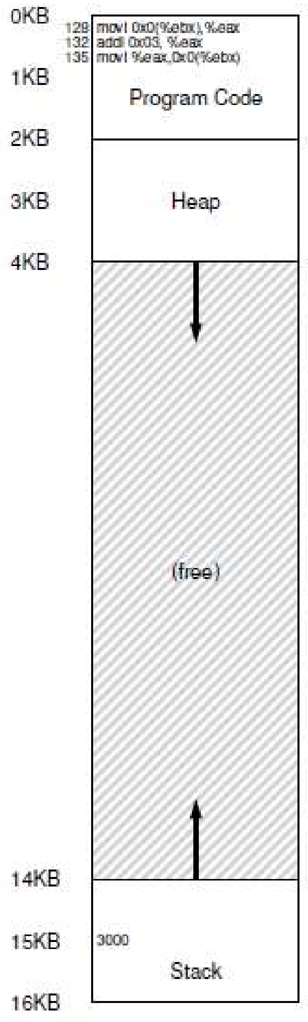
```
void func() {
    int x = 3000;
    x = x + 3;
```

- It is compiled as follows

virtual addr

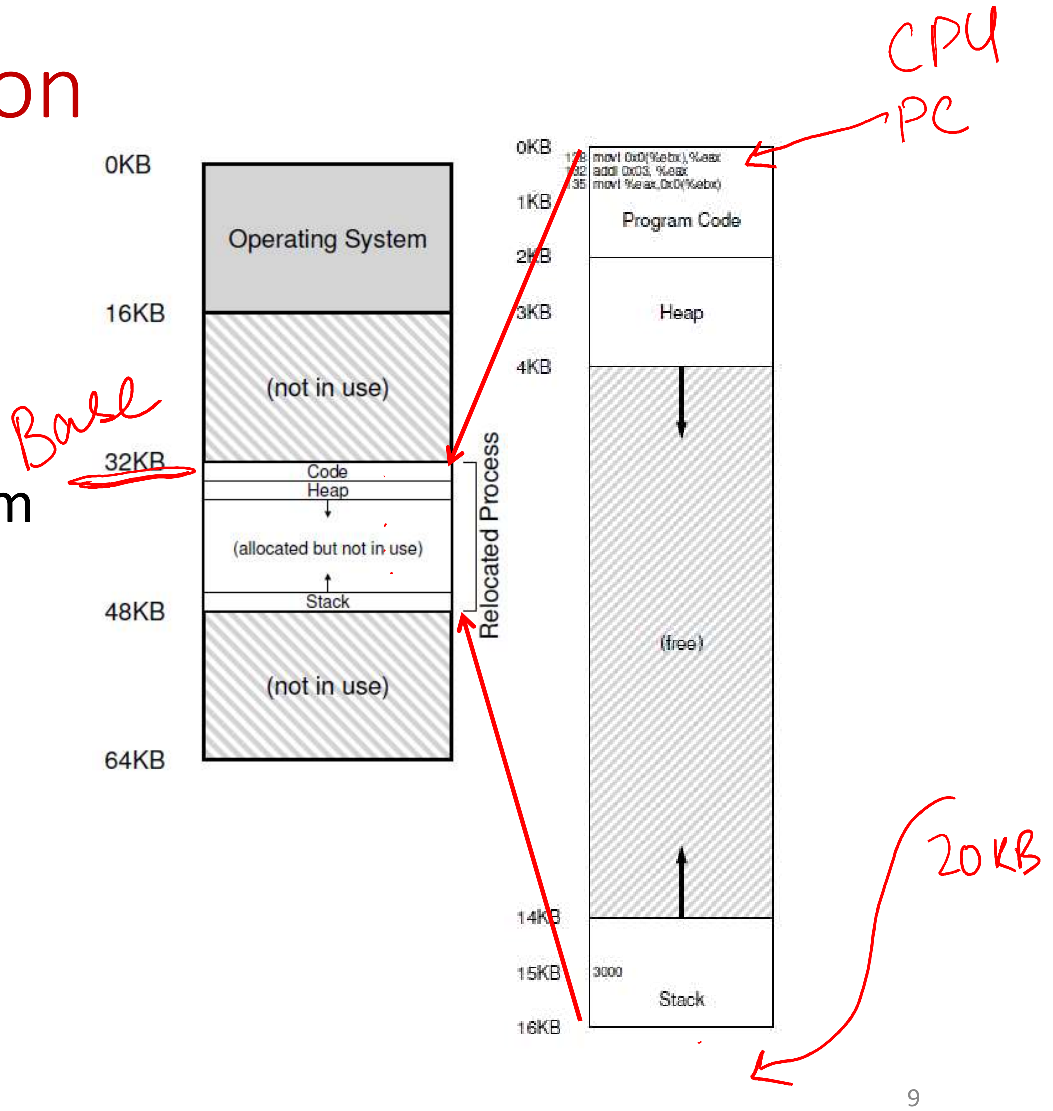
```
128: movl 0x0(%ebx), %eax    ;load 0+ebx into eax
132: addl $0x03, %eax        ;add 3 to eax register
135: movl %eax, 0x0(%ebx)    ;store eax back to mem
```

- Virtual address space is setup by OS during process creation



Address Translation

- Suppose OS places entire memory image in one chunk, starting at physical address 32KB
- Need the following translation from VA to PA
  - 128 to 32896 (32KB + 128)
  - 1KB to 33 KB
  - 20KB? Error!



```
128 movl 0x0(%ebx),%eax
132 addl 0x03, %eax
135 movl %eax,0x0(%ebx)
```

Who performs address translation?

- In this simple example, OS tells the MMU hardware the base (starting address) and bound (total size of process) values
- Memory hardware Memory Management Unit (MMU) calculates PA from VA

33KB 1KB 32 KB

```
physical address = virtual address + base
```

20 KB

- MMU also checks if address is beyond bound
- OS is not involved in every translation

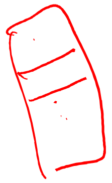
trap

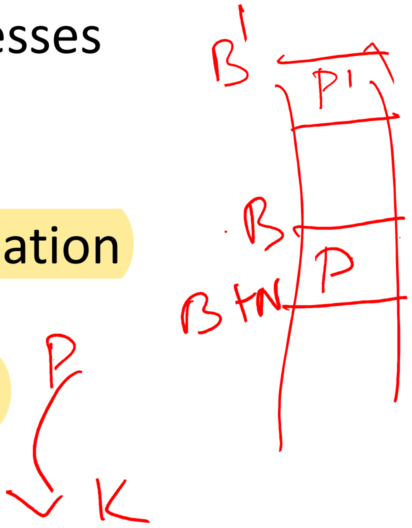
$$\begin{tikzcd} CPU \arrow[r, "x"'] & MMU \arrow[r, "B+x"'] & RAM \arrow[ll, bend left, "data"] \end{tikzcd}$$

Role of OS vs MMU

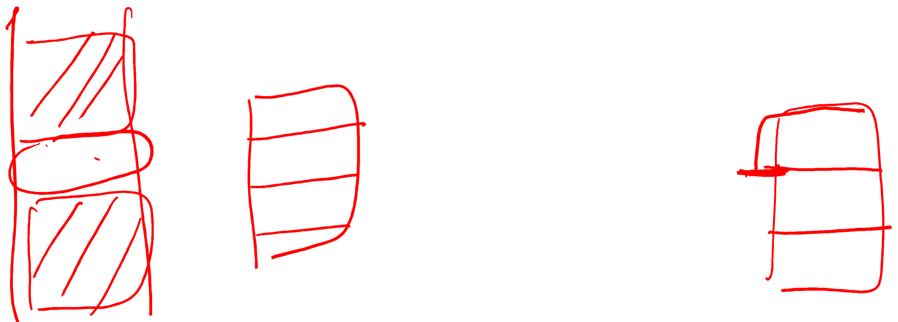
base, bound

- OS allocates memory, builds translation information of process
  - But OS does not do the actual address translation on every memory access
  - Why? Once user code starts running on CPU, OS is out of the picture (until a trap)
- When process is switched in, translation information is provided to MMU
- CPU runs process code, accesses code/data at virtual addresses
  - Virtual addresses translated to physical addresses by MMU
  - Actual physical memory is accessed using physical addresses
- MMU raises a trap if there is any error in the address translation
  - CPU executes trap instruction, OS code runs to handle the error
- OS gives new information to MMU on every context switch





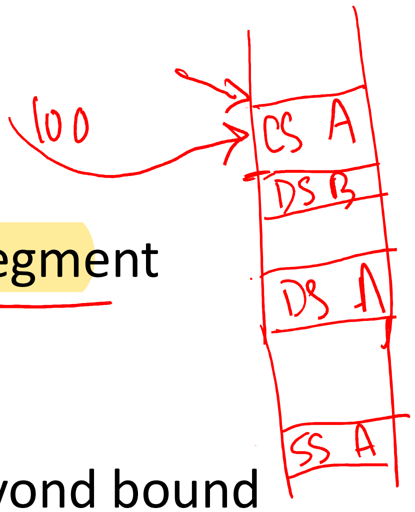
Segmentation

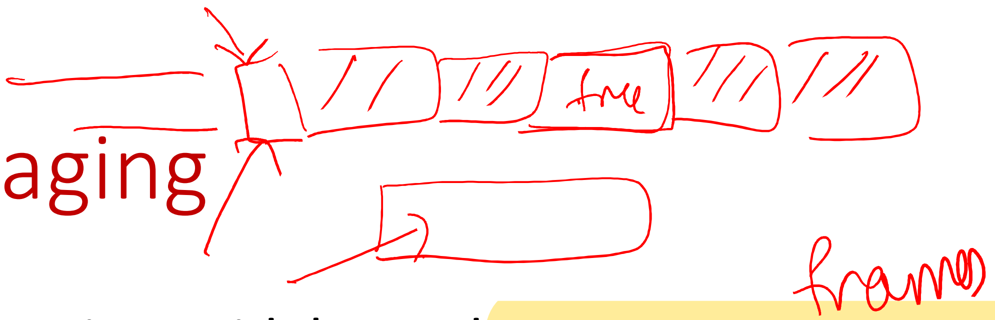


- Older way of memory management, generalized base and bounds
- Each segment of the program (code, data, stack,..) is placed separately in memory at a different base
    - Every segment has a separate base and bound
- Virtual address = segment identifier : offset within segment
- Physical address = base address of segment + offset within segment
    - Bound of a segment checked for incorrect access
- Multiple base, bound values stored in MMU for translation
- MMU throws a segmentation fault if a segment accessed beyond bound
    - Program fault, traps to OS to handle error, may terminate process

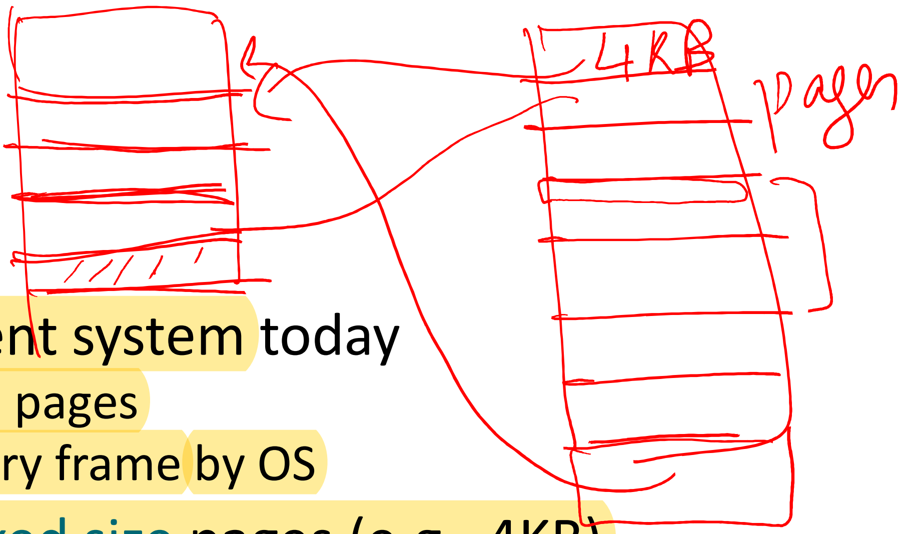
PC 100

CS : 100

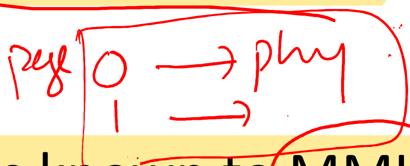




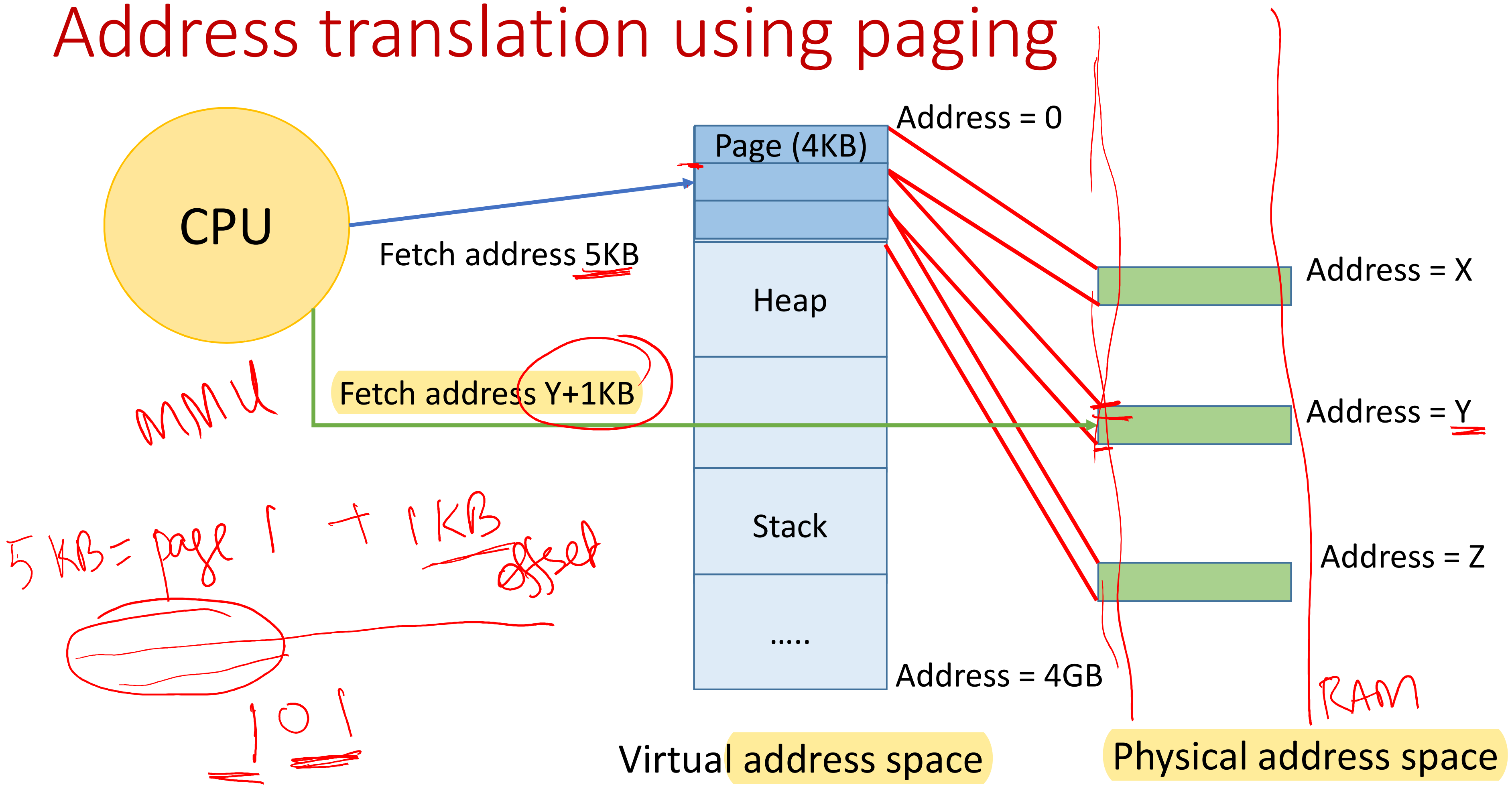
Paging

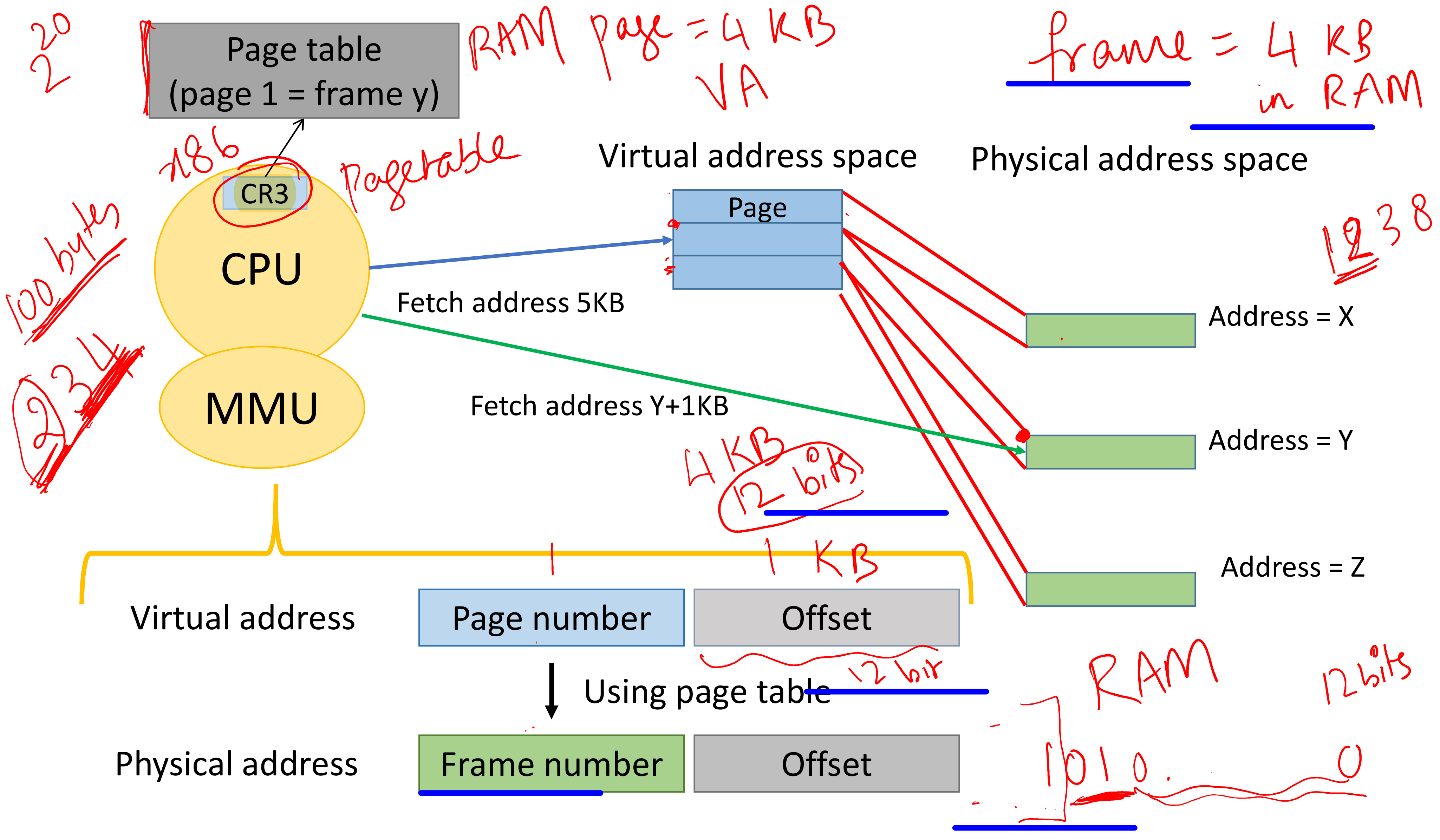


- Paging: widely used memory management system today
  - Virtual address space divided into fixed size pages
  - Each page is assigned a free physical memory frame by OS
- Memory allocation is at granularity of fixed size pages (e.g., 4KB)
  - Avoids external fragmentation (no wastage of space between pages)
  - Internal fragmentation (space may be wasted inside partially filled page)
- Page table maps logical page numbers to physical frame numbers
  - One page table per process
  - Maintained by OS, part of PCB of process
- Location of page table of currently running process known to MMU
  - Written into special CPU register, updated on context switch/page table change

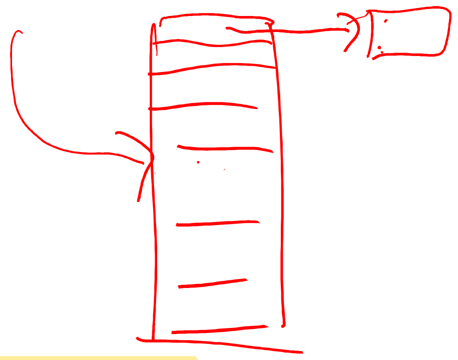


Address translation using paging



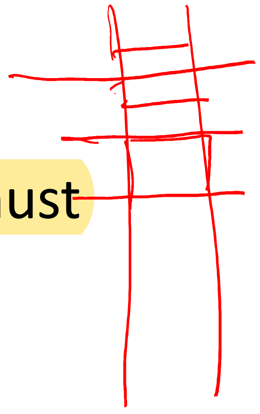

Page table
(page 1 = frame y)
CR3
CPU
MMU
Virtual address space
Page
Physical address space
Fetch address 5KB
Fetch address Y+1KB
Address = X
Address = Y
Address = Z
Virtual address
Page number
Offset
Using page table
Physical address
Frame number
Offset
2^{20}
x86
Pagetable
100 bytes
234
RAM page = 4 KB
VA
frame = 4 KB
in RAM
1238
4 KB
12 bits
1 KB
12 bit
RAM
12 bits
1010 . 0

Address translation using paging



- Address translation performed by MMU
  - Virtual address accessed by CPU = page number | offset within page
  - Find frame number corresponding to page number by looking up page table
  - Physical address = physical frame number | offset within page
- Example: 32-bit CPU, 4 KB pages
  - 32 bit virtual address = 20 bit page number | 12 bit offset within page
  - Page table maps 2^20 pages to physical frame numbers
  - Physical address = Physical frame number | 12 bit offset within page
- Overhead of paging: before doing actual memory access, MMU must do extra memory access for page table mapping
  - How to avoid this?

$\log_2 4KB = 12$



$$\begin{tikzcd}
CPU \arrow[r, "V"] & MMU \arrow[r, "P"] \arrow[dr, bend right] & RAM \\
 & & PT
\end{tikzcd}$$

$$\begin{tikzcd}
\text{CPU} \atop \boxed{\text{caches}} \arrow[r, "V"] & \text{MMU} \atop (\underline{\text{TLB}})(V \to P) \arrow[r, "P"] \arrow[dr, bend right] & \cancel{\text{PT}}\ \text{RAM} \\
 & & \text{P.T}
\end{tikzcd}$$

Translation Lookaside Buffer (TLB)

- Overhead of memory translation: every memory access preceded by extra memory accesses to read page table
- To reduce this overhead, MMU caches the most recent translations in translation lookaside buffer (TLB)
  - Small cache within MMU to store page number to frame number mappings
  - LRU policy to evict entries if TLB is full (locality of reference)

$VA \to PA$ $\qquad V \to P$

- TLB only caches address translations, not actual memory contents
  - Different from CPU caches that cache actual memory contents
  - If TLB hit, physical address is ready, fetch memory contents in one memory access
  - If TLB miss, extra memory access for page table access also needed
- TLB flush on context switch: mappings cached in TLB change

$$\begin{tikzcd}
\text{CPU} \arrow[d] \arrow[r, "V", "(\text{Caches})"'] & \text{MMU} \arrow[r, "P", "V(\text{TLB})P"'] \arrow[rd] & \text{RAM} \sim 100\text{ ns} \\
\sim \text{ns } 10^{-9} & & \text{P.T}
\end{tikzcd}$$

What happens on a memory access?

- CPU has requested data (or instruction) at a certain memory address
  - If requested address not in CPU cache, CPU must fetch data from main memory
  - CPU knows only virtual address of instruction or data required
  - MMU looks up TLB to find frame number corresponding to page number
  - If TLB hit, physical address is found, main memory is accessed to fetch data
  - If TLB miss, MMU first accesses page table in main memory, computes physical address (translation added to TLB cache), then accesses main memory again to fetch data
  - Fetched data added to CPU caches for future use
- High CPU cache hit rates and high TLB hit rates are important for good performance of the system