Introduction to memory management

Mythili Vutukuru
CSE, IIT Bombay

Memory management in OS

- When program is run, memory image of process is created by OS
  - Code + compile-time data from executable loaded into main memory
  - Extra memory allocated for stack and heap
- CPU begins executing process
- How does CPU locate code/data in memory? Using memory addresses

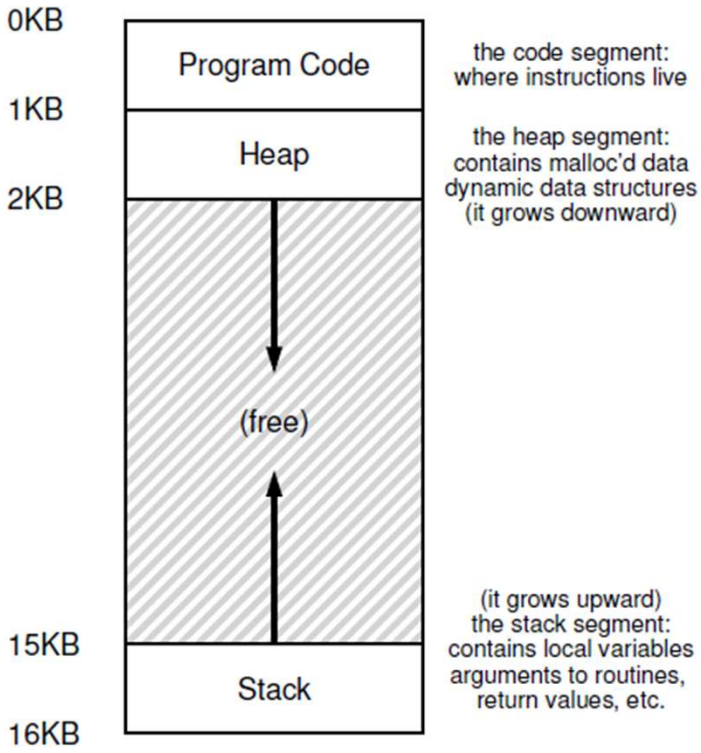


Image credit: OSTEP

Memory access using physical addressing

- CPU can use the actual physical address (byte # in RAM) of the instruction or data to fetch it
- Not very convenient or practical
  - How does compiler know the physical addresses it must assign to code/data in the executable at compile time?
  - What if we need to move the memory image to another location in RAM?
- Modern systems use the concept of virtual addressing

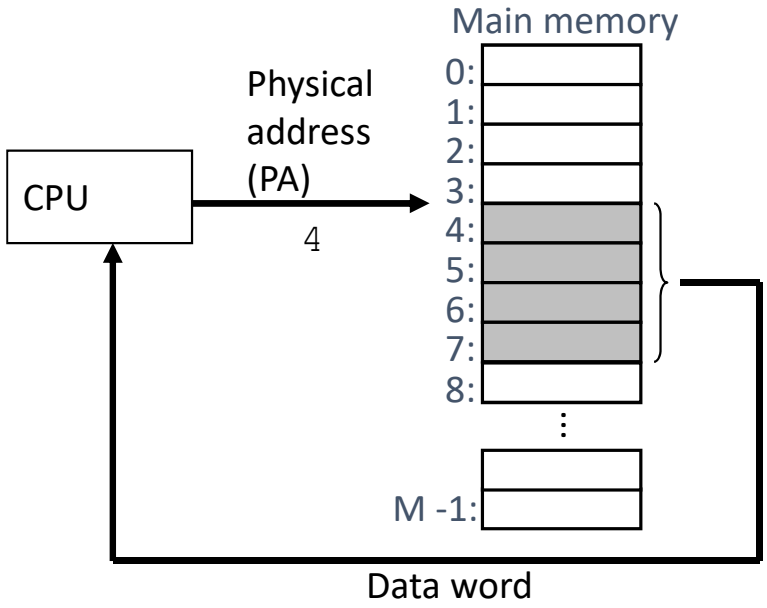


Image credit: CSAPP

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 value
- Max value depends on #bits available to address memory
  - 2^32 = 4GB in 32-bit machines
- Virtual address space contains all code/data that a process can access
- Addresses in CPU registers and pointer variables = virtual addresses
- CPU issues loads and stores to virtual addresses

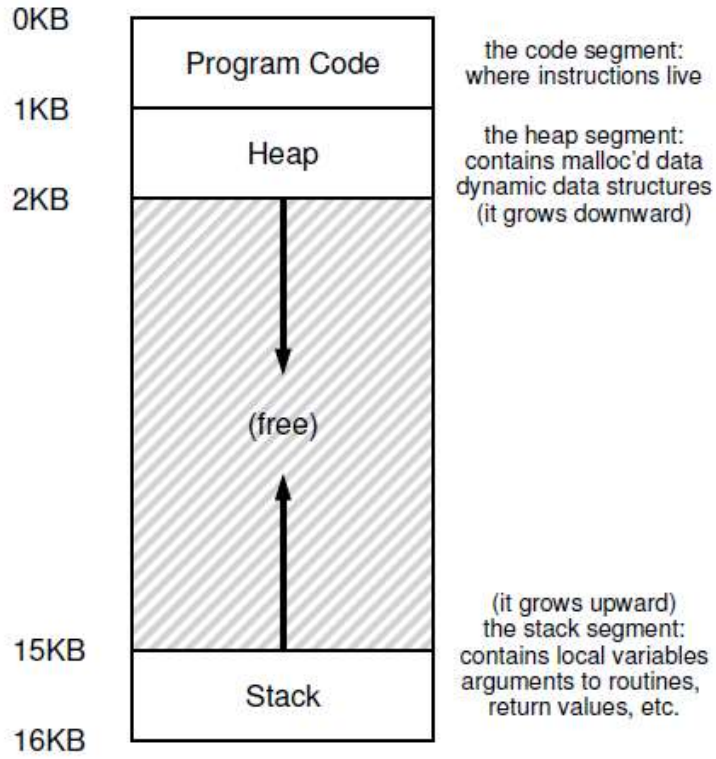


Figure 13.3: **An Example Address Space**

Image credit: OSTEP

How is actual memory reached?

- Physical address space: actual physical memory in RAM
- Every virtual address space is mapped to physical addresses in memory by OS
- On every memory access, virtual address (VA) translated to physical address (PA) by special hardware called Memory Management Unit (MMU)

Every memory fetch by CPU is to be passed from MMU which translates the VA to PA.

- OS allocates physical memory to a process, has translation information, provides it to MMU

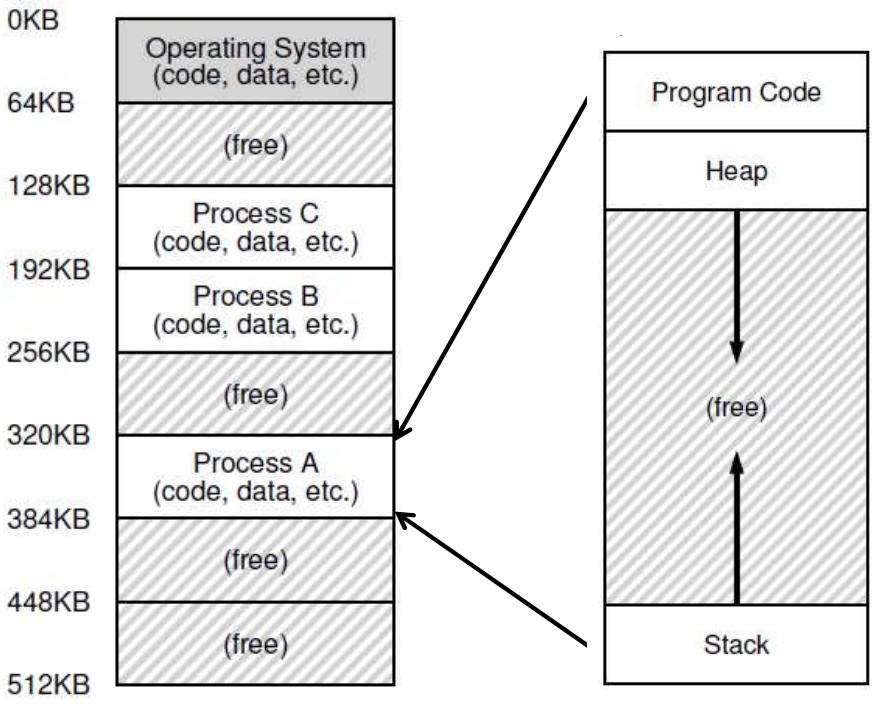


Image credit: OSTEP

Memory access using virtual addressing

- Code+data in memory image is assigned virtual addresses starting from 0 (by compiler and OS)
- CPU fetches code/data using virtual addresses, MMU translates to physical addresses (using info provided by OS)

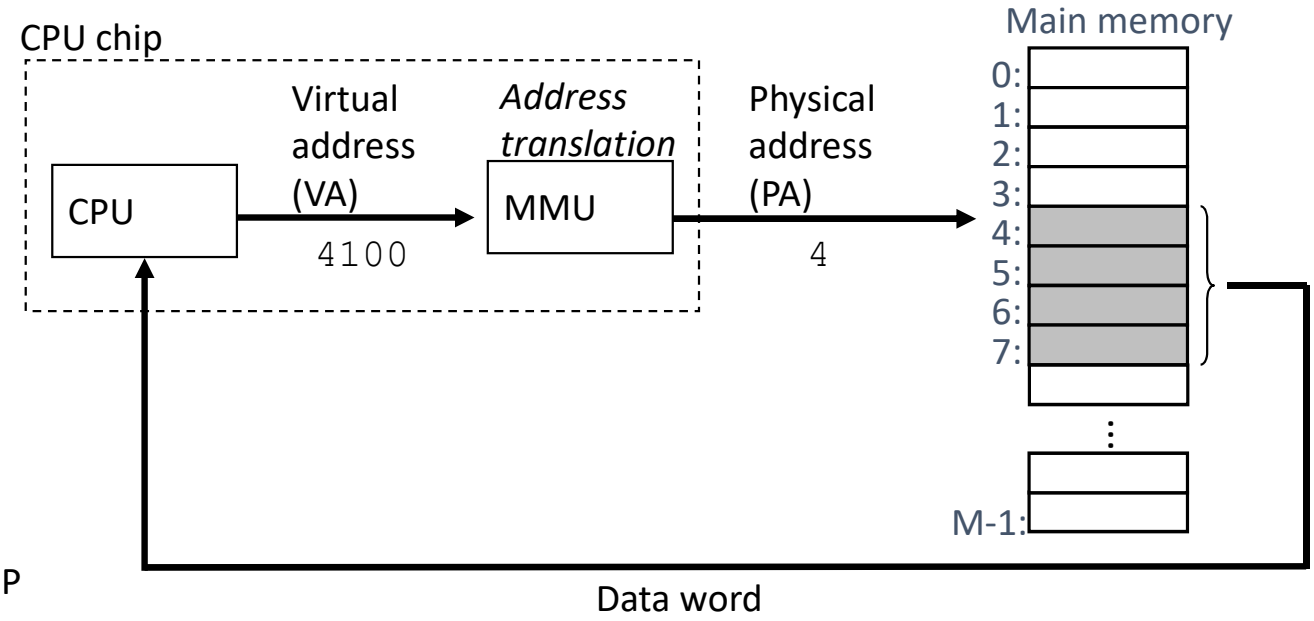
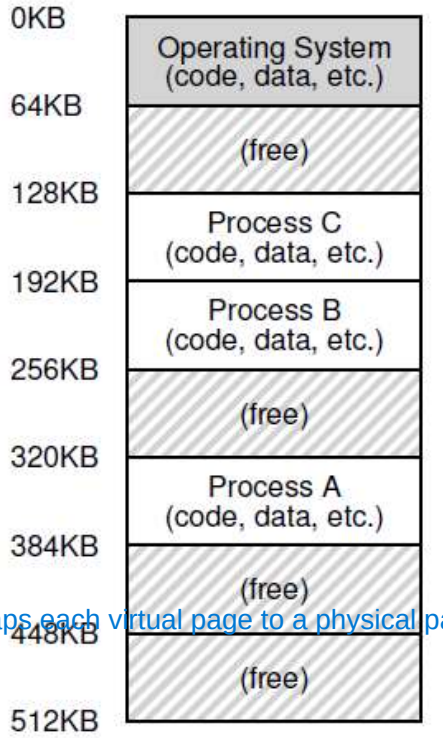


Image credit: CSAPP

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 allocation can be non-contiguous
- Need to hide this complexity from user
- Also, physical address not known at compile time



The OS sets up a page table at runtime. When a process is loaded, the OS allocates physical pages and maps each virtual page to a physical page via this table.

The compiler only deals with virtual addresses; the actual physical mapping is determined when the program runs.

Figure 13.2: **Three Processes: Sharing Memory**

Image credit: OSTEP

Base and bound

- How are virtual address spaces mapped to physical memory?
- Simplest form of memory management: base and bound
  - Place 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
- OS provides base and bound to MMU for translation/error checking
- When CPU access a VA, MMU computes PA = VA + base, physical memory accessed with PA

A simple example

- Consider a simple C function

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

- It is compiled as follows

```
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

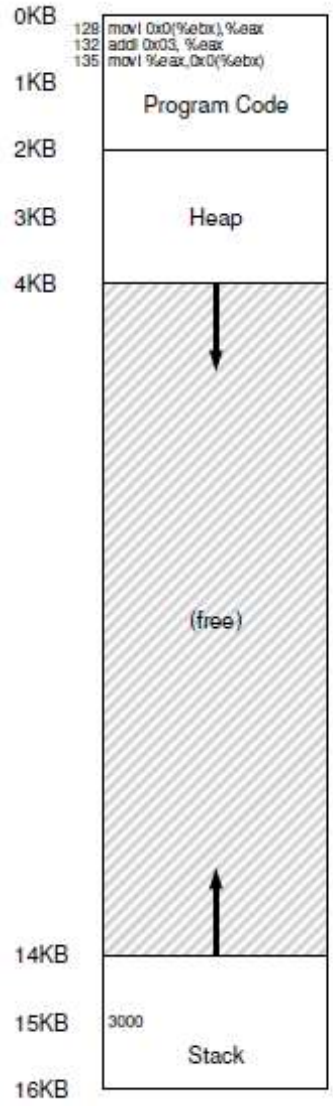


Image credit: OSTEP

Example: address translation

- Suppose OS places entire memory image in one chunk, starting at physical address 32KB
- OS indicates base and bound to MMU
- MMU performs the following translation from VA to PA

In reality offset is extracted and used as an index

- PA = VA + base
  - VA = 128, PA = 32896 (32KB + 128)
  - VA = 1KB, PA = 33 KB
  - VA = 20KB? PA = ???
- MMU raises trap when address out of bound

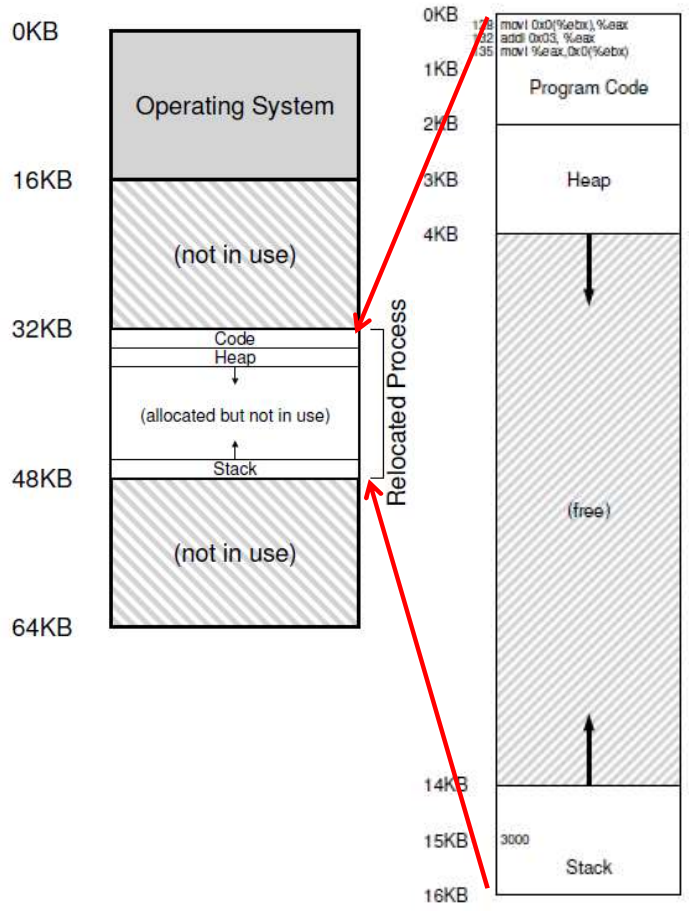


Image credit: OSTEP

Role of OS vs MMU

- 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

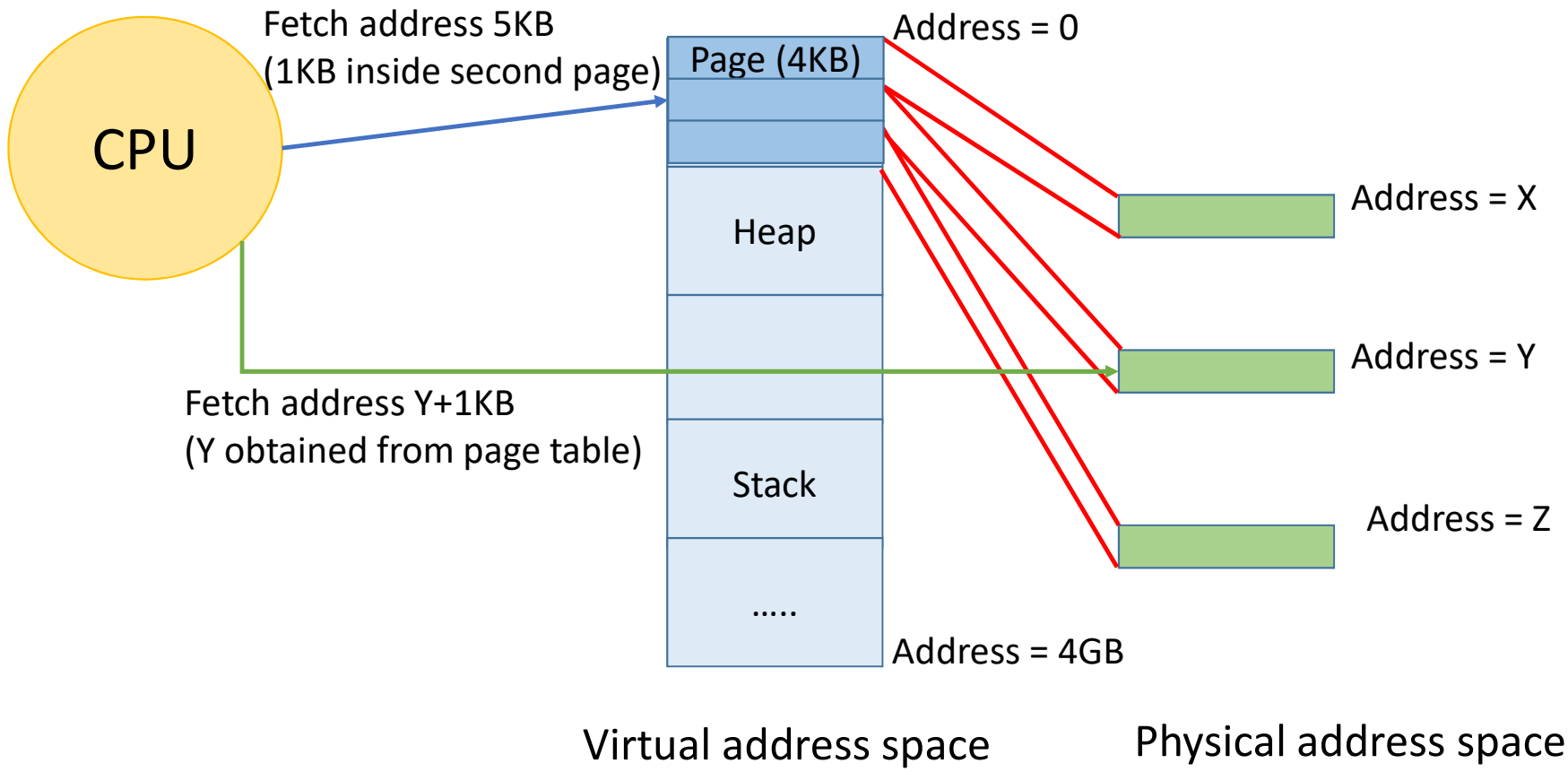
Segmentation Older way to manage the memory.

- 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

Paging

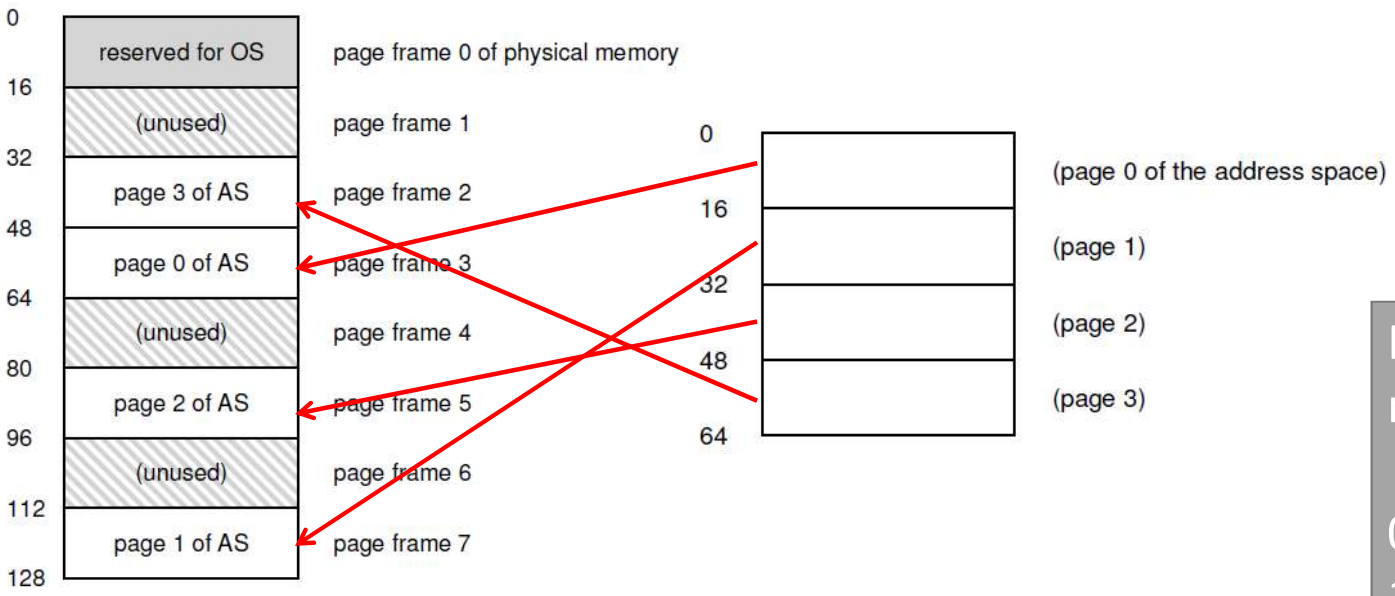
- Widely used memory management system today
- Virtual address space divided into fixed size pages
- Each page is assigned a free physical frame by OS
- Memory allocation is at granularity of fixed size pages (e.g., 4KB)
- Why paging? Avoids external fragmentation
  - No wastage of space due to gaps between allocated and free memory
  - Internal fragmentation may be there (space wasted inside partially filled page)
- Disadvantage: internal fragmentation (partially filled pages)
- Page table maps logical page numbers to physical frame numbers

Paging



It is not necessary that entire entry of 4kb is filled leading to space wastage.

Example of paging



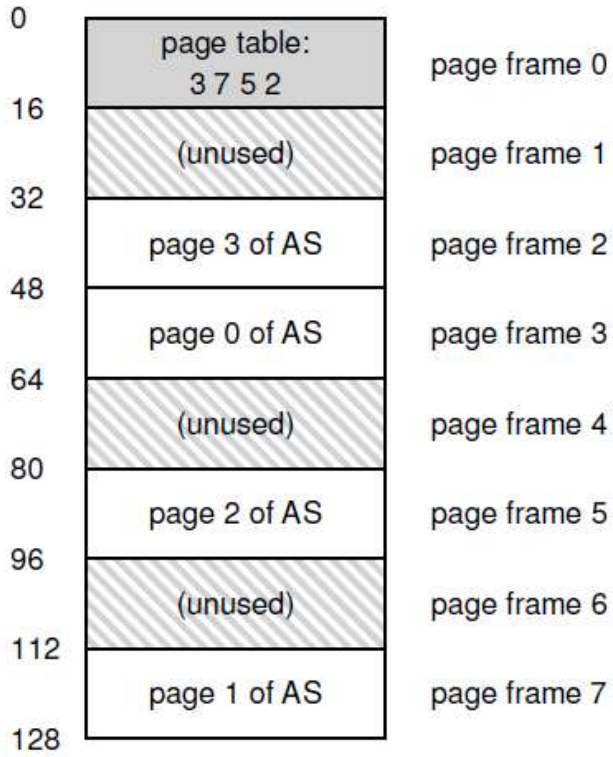
Page to frame mappings:

0 → 3
1 → 7
2 → 5
3 → 2

Image credit: OSTEP

Page table

- Per process data structure to translate virtual address (VA) to physical address (PA)
- Stores frame numbers for all pages of a process in array
  - [3 7 5 2] corresponding to pages 0 to 3 of the process
- Part of OS memory (in PCB)
- MMU has access to page table of current process, uses it for address translation



View of physical memory

Image credit: OSTEP

Address translation using paging

- Address translation performed by MMU using page table
- Most significant bits (MSB) of VA give virtual page number, least significant bits (LSB) give offset within page
- Page table maps virtual page number (VPN) to physical frame number (PFN)
- MMU maps VPN to PFN, adds offset to get PA
- Location of page table of currently running process known to MMU
  - Written into special CPU register by the OS, updated on every context switch/page table change

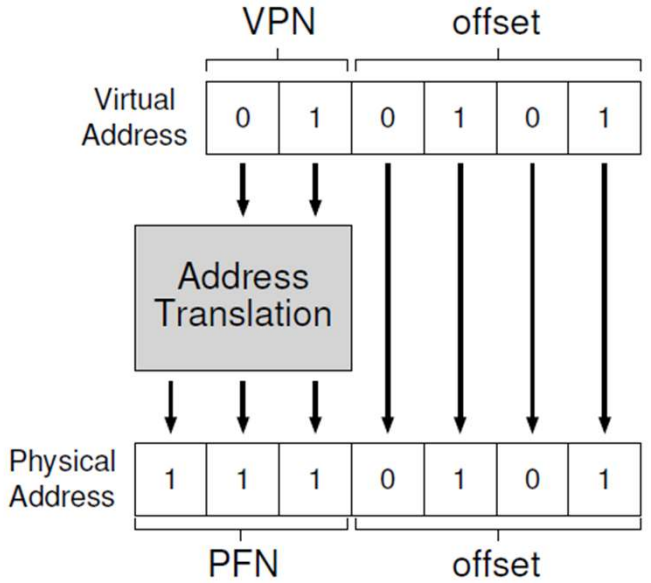


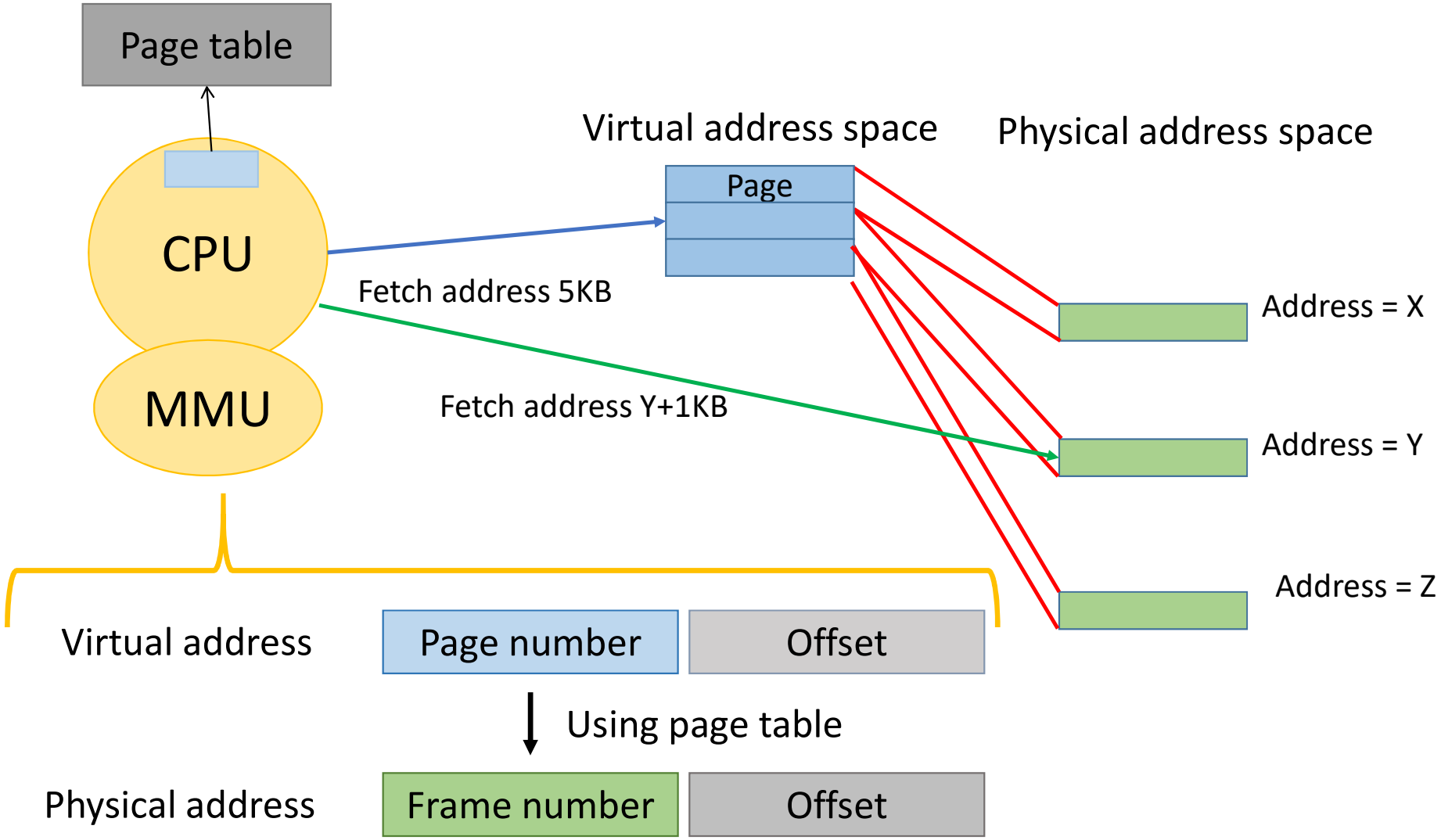
Figure 18.3: **The Address Translation Process**

Image credit: OSTEP

Example: 32-bit systems

1K = 2^10 = 1024
1M = 2^20 = 1024*1024
1G = 2^30 = 1024 * 1024 * 1024
B = byte, b = bit

- 32 bit virtual addresses, so virtual address space size is 2^32 = 4GB
- Typical page size = 4KB = 2^12 bytes
- Offset within page needs 12 bits
- 32 bit VA = 20 bit VPN + 12 bit offset
- Number of pages in address space of process = 4GB / 4KB = 1M
- Size of page table array is 1M entries per process!


Page table
CPU
MMU
Virtual address space
Physical address space
Page
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

Page table structure

- Page table is an array, where i-th entry contains the information (physical frame number etc) of the i-th page of the process
- Page table has entries for all pages in address space, even those for which there is no physical frame number
  - It is an array with fixed number of entries, not a dynamic data structure
  - Why this design? Why not a dynamic data structure whose size depends on number of used pages?
- Page table structure fixed because the logic to traverse the page table is baked into MMU hardware

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)
- TLB only caches page table entries (VPN → PFN mappings), not actual memory contents
  - Different from CPU caches that cache actual memory contents
- If TLB hit, fetch memory contents in one memory access
- If TLB miss, MMU must perform extra memory access for page table access ("page table walk")
- TLB flush on context switch: mappings cached in TLB change

TLB hit

1. CPU accesses virtual address
2. MMU looks up page number in TLB
3. If TLB hit, page table entry is available, physical address computed
4. CPU directly accesses required code/data using physical address

Data accessed in one memory visit.

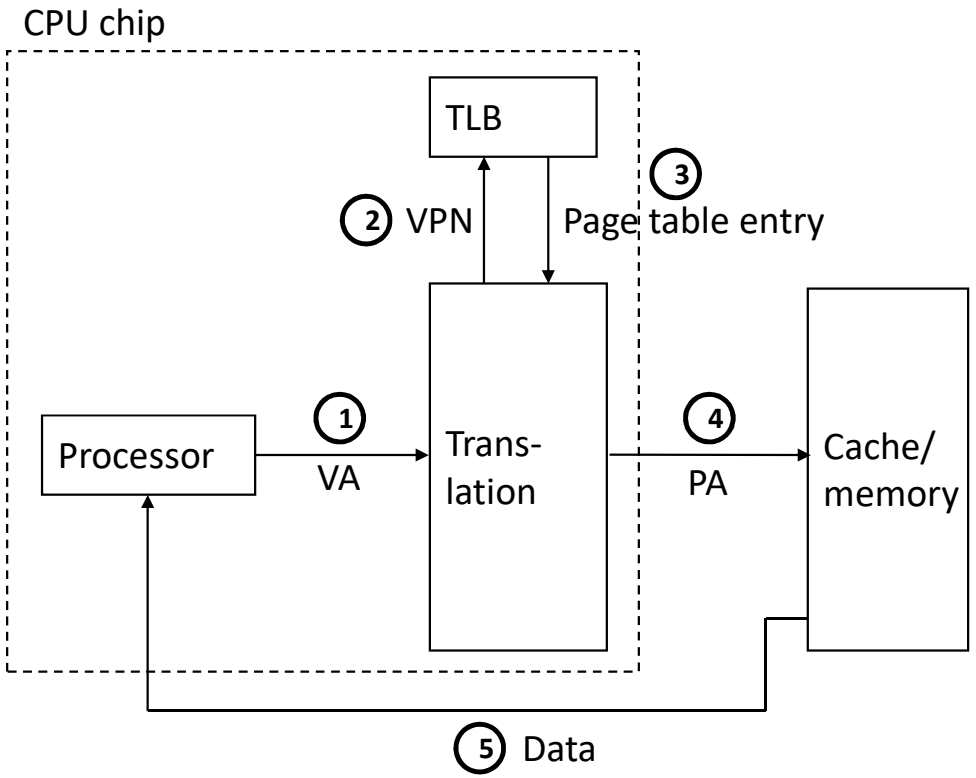
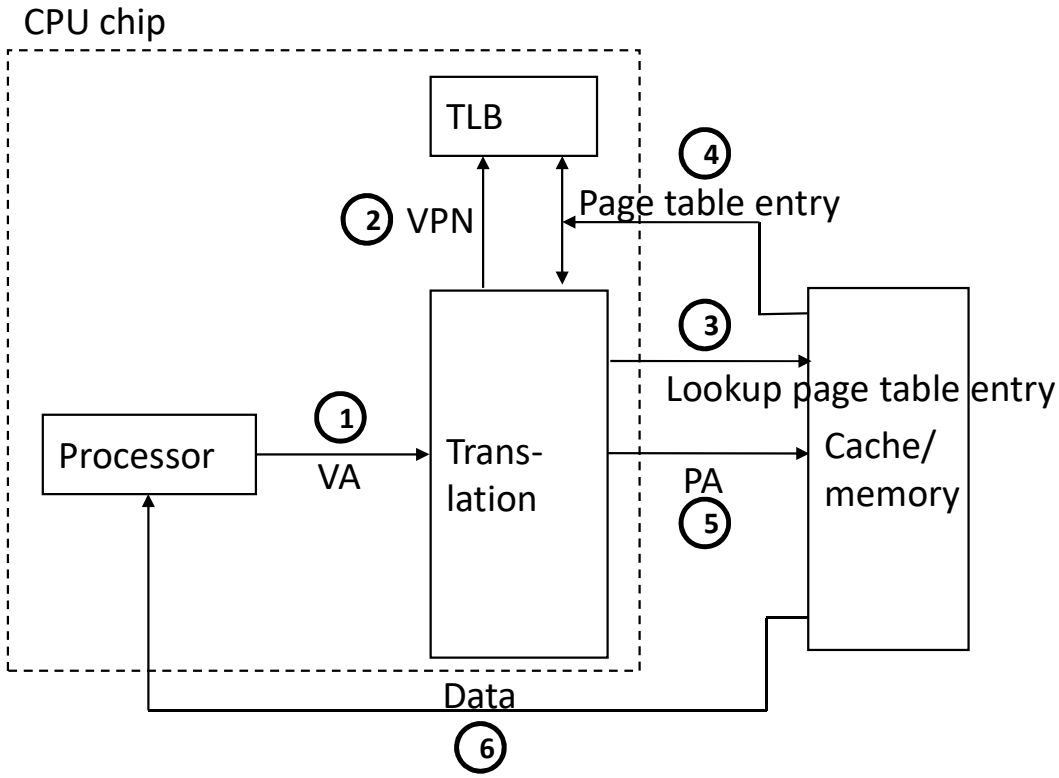


Image credit: CSAPP

TLB miss

1. CPU accesses virtual address
2. MMU looks up page number in TLB, cannot find entry
3. MMU looks up page table in memory to find page table entry
4. Page table entry populated in TLB for future use
5. MMU computes physical address using which CPU accesses main memory



Two times memory access when TLB miss happened

Image credit: CSAPP

Putting it all together: 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, then accesses main memory again to fetch data
  - Fetched page table entries and data are populated in TLB / caches
- High CPU cache hit rates and high TLB hit rates are important for good performance of the system