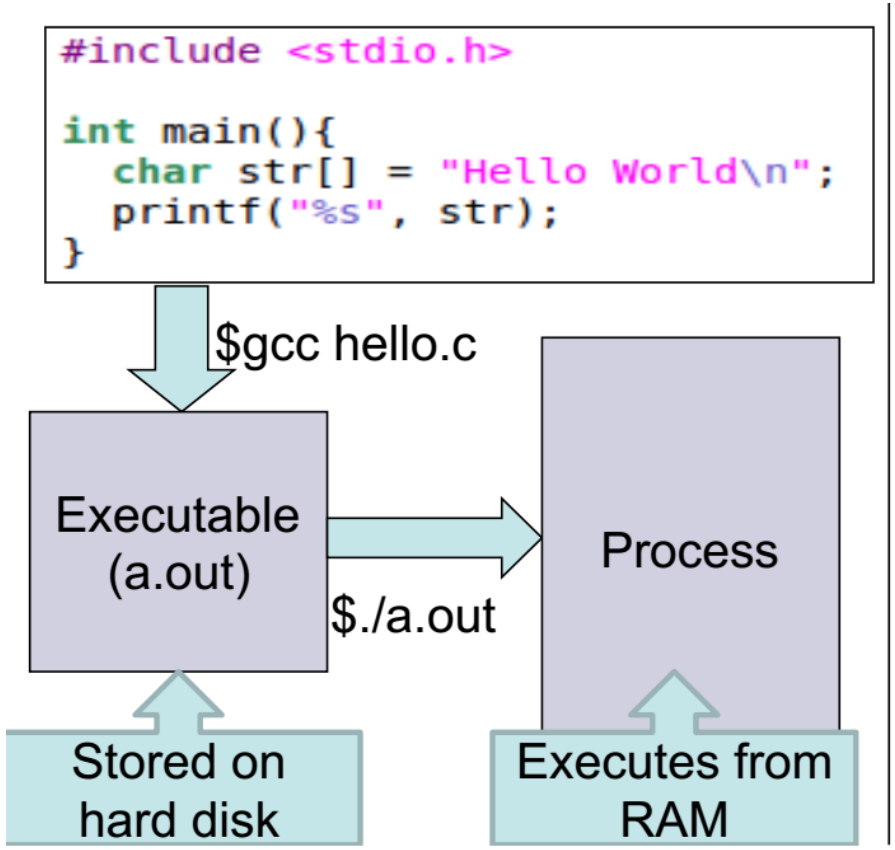
CS304 Operating Systems

DR GAYATHRI ANANTHANARAYANAN

gayathri@iitdh.ac.in

Materials in these slides have been borrowed from textbooks and existing operating systems courses

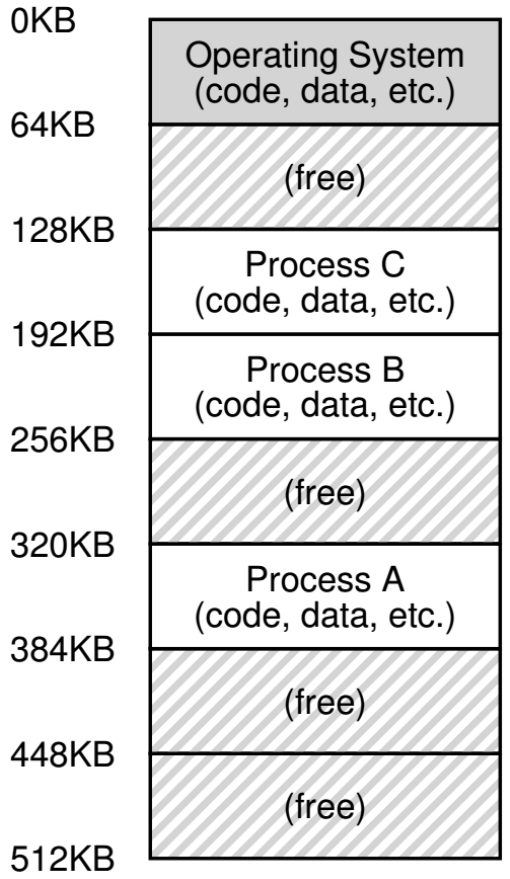
Abstraction: (Virtual) Address Space



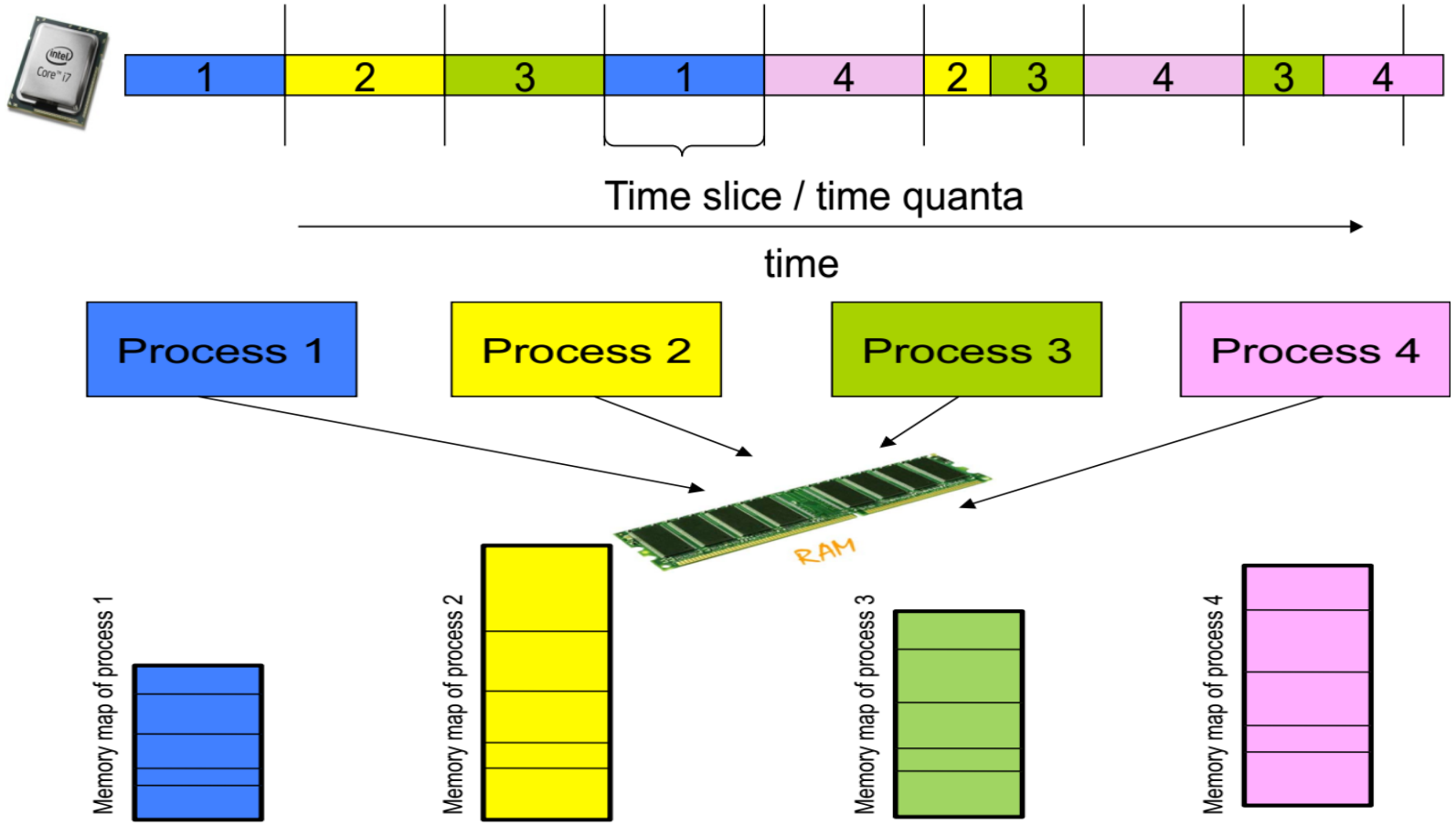
- Process
  - A program in execution
  - Present in the RAM
  - Comprises of
    - Executable instructions
    - Stack
    - Heap
    - State in the OS (in kernel)
  - State contains : registers, list of open files, related processes, etc.

Virtual Memory - Introduction

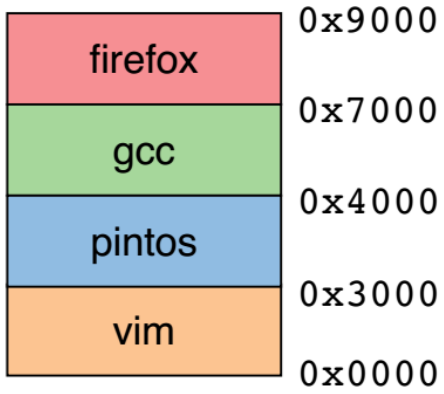
- Because real view of memory is messy!
- Earlier, memory had only code of one running Process (and OS code)
- Now, multiple active processes timeshare CPU
  - Memory of many processes must be in memory
  - Non-contiguous too
- Need to hide this complexity from user



Multitasking



- **Rewind to the days of "second-generation" computers**
  - Programs use physical addresses directly
  - OS loads job, runs it, unloads it
- **Multiprogramming changes all of this**
  - Want multiple processes in memory at once
- **Consider multiprogramming on physical memory**
  - What happens if pintos needs to expand?
  - If vim needs more memory than is on the machine?
  - If pintos has an error and writes to address 0x7100?
  - When does gcc have to know it will run at 0x4000?
  - What if vim isn't using its memory?



Issues in Sharing Physical Memory

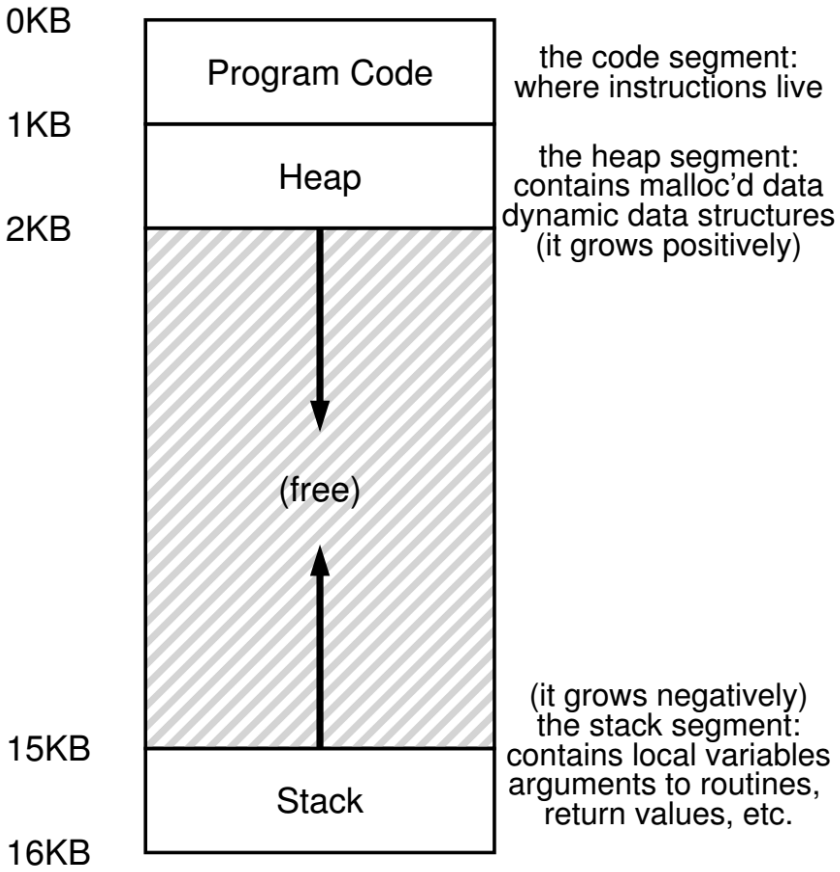
- **Protection**
  - A bug in one process can corrupt memory in another
  - Must somehow prevent process *A* from trashing *B*'s memory
  - Also prevent *A* from even observing B's memory (ssh-agent)
- **Transparency**
  - A process shouldn't require particular physical memory bits
  - Yet processes often require large amounts of contiguous memory (for stack, large data structures, etc.)
- **Resource exhaustion**
  - Programmers typically assume machine has "enough" memory
  - Sum of sizes of all processes often greater than physical 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

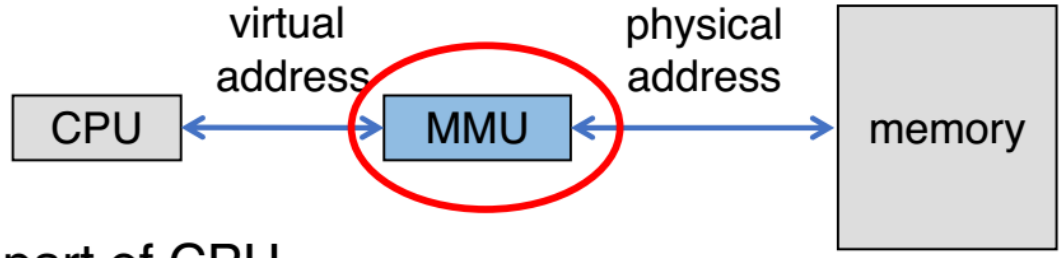
(dynamic allocations)and stack (used during function calls)

- Stack and heap grow during runtime
- CPU issues loads and stores to virtual addresses



Definitions

- **Programs load/store to virtual addresses**
- **Actual memory uses physical addresses**
- **VM Hardware is Memory Management Unit (MMU)**



  - Usually part of CPU
    - Configured through privileged instructions (e.g., load bound reg)
  - Translates from virtual to physical addresses
  - Gives per-process view of memory called address space

How is actual memory reached?

- Address translation from virtual addresses (VA) to physical addresses (PA)

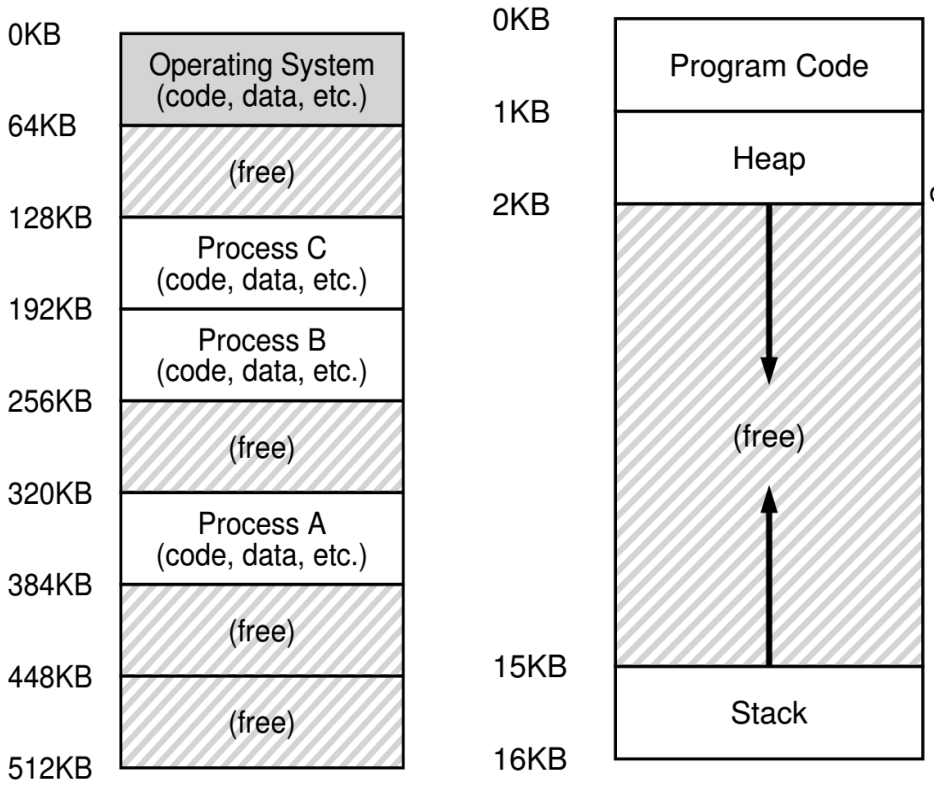
–CPU issues loads/stores to VA but memory hardware accesses PA

- OS allocates memory and tracks location of processes

- Translation done by memory hardware called

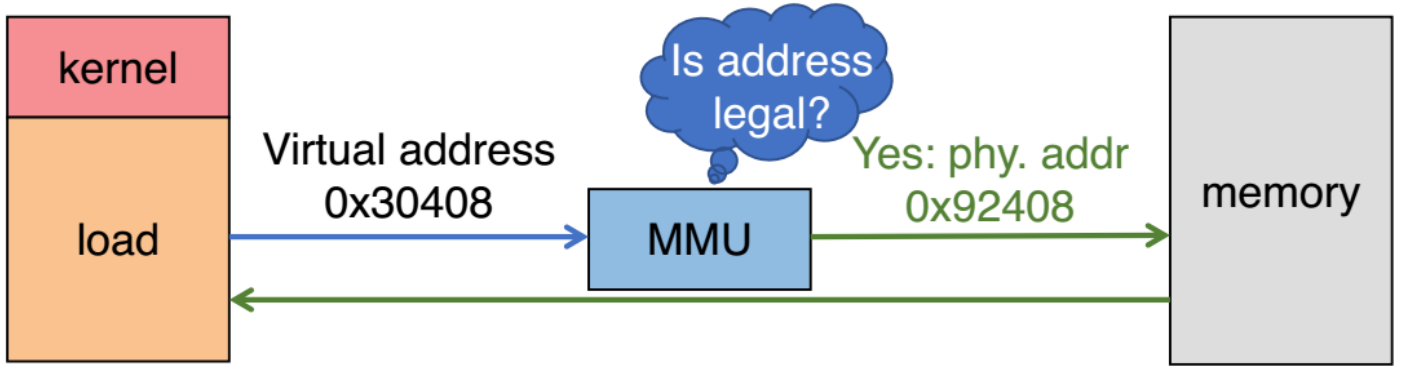
Memory Management Unit (MMU)

–OS makes the necessary information available



Goals of memory virtualization

- **Transparency:** user programs should not be aware of the messy details
- **Efficiency:** minimize overhead and wastage in terms of memory space and access time
- **Isolation and protection:** a user process should not be able to access anything outside its address space



- **Give each program its own virtual address space**
  - At runtime, Memory-Management Unit relocates each load/store
  - Application doesn't see physical memory addresses
- **Enforce protection**
  - Prevent one app from messing with another's memory
- **And allow programs to see more memory than exists**
  - Somehow relocate some memory accesses to disk

Virtual Memory Advantages

- **Can re-locate program while running**
  - Run partially in memory, partially on disk
- **Most of a process's memory may be idle (80/20 rule)**
  - Write idle parts to disk until needed
  - Let other processes use memory of idle part
  - Like CPU virtualization: when process not using CPU, switch (Not using a memory region? switch it to another process)
- **Challenge: VM = extra layer, could be slow**

How can a user allocate memory?

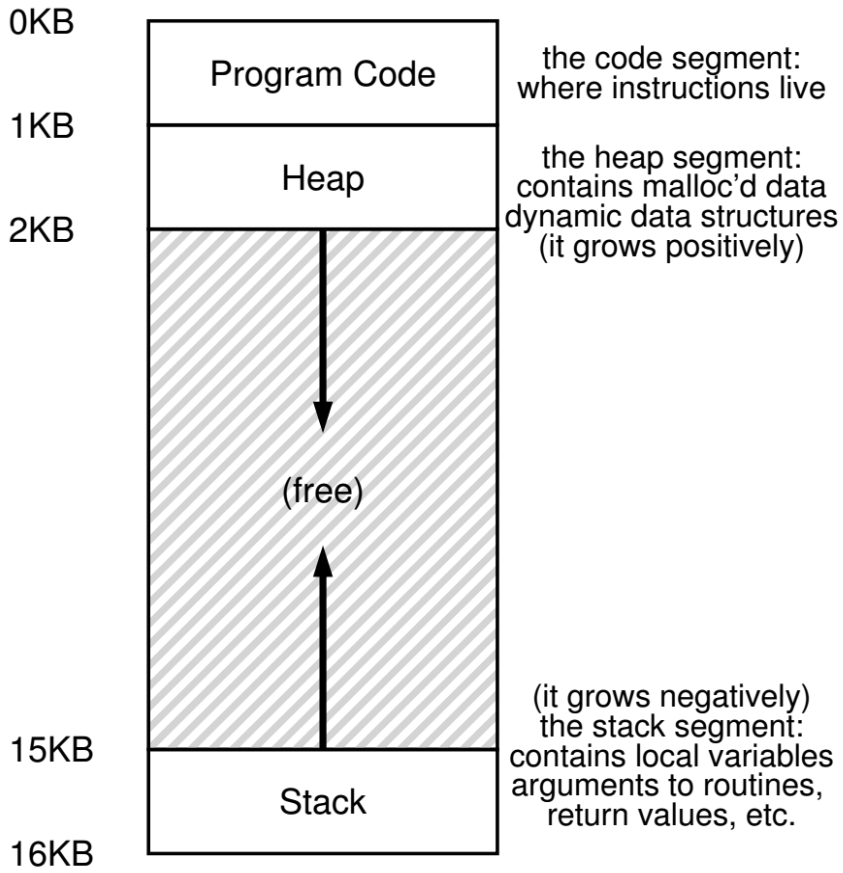
OS allocates a set of pages to the memory image of the process

- Within this image

–Static/global variables are allocated in the executable

–Local variables of a function on stack

–Dynamic allocation with malloc on the heap



Memory allocation system calls

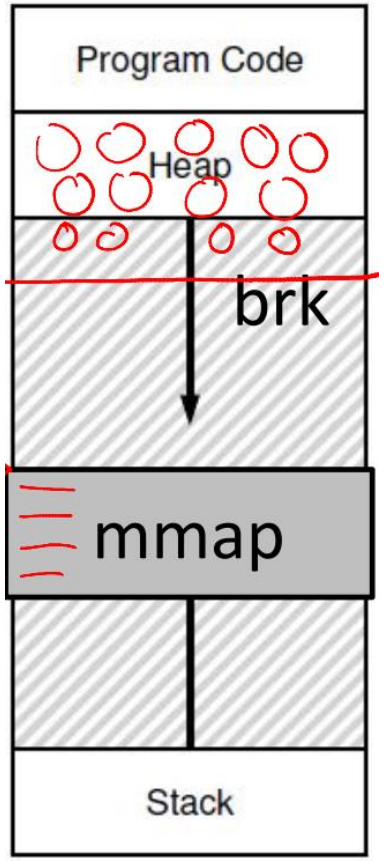
- **`malloc`** implemented by C library

–Algorithms for efficient memory allocation and free space management

- To grow heap,libc uses the **`brk/sbrk`** system call

- A program can also allocate a page sized memory using the **`mmap()`** system call

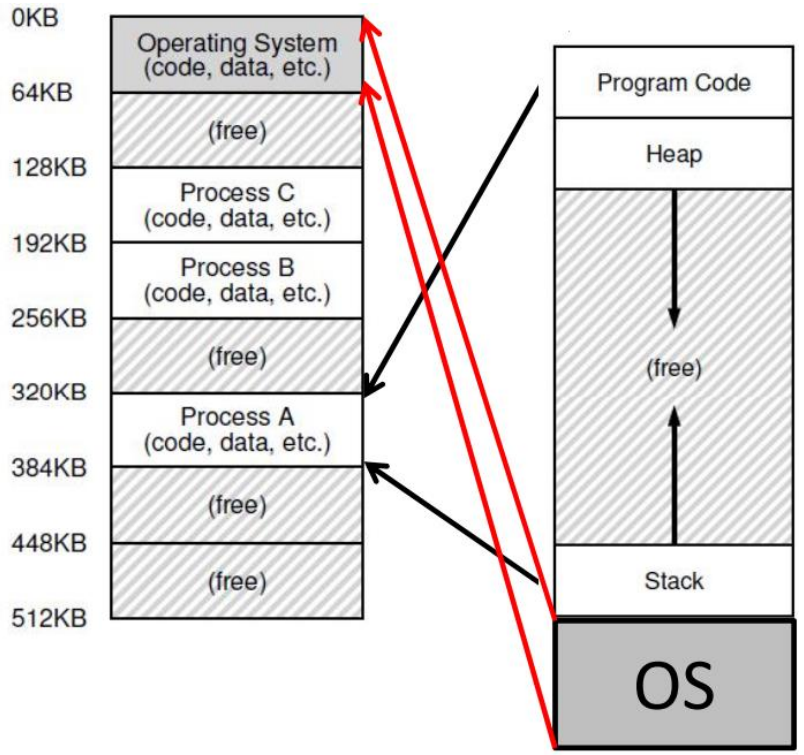
–Gets "anonymous" page from OS



A subtle point: what is the address space of the OS?

OS is not a separate process with its own address space

- Instead, OS code is part of the address space of every process
- A process sees OS as part of its code (e.g., library)
- Page tables map the OS addresses to OS code



How does the OS allocate memory for itself?

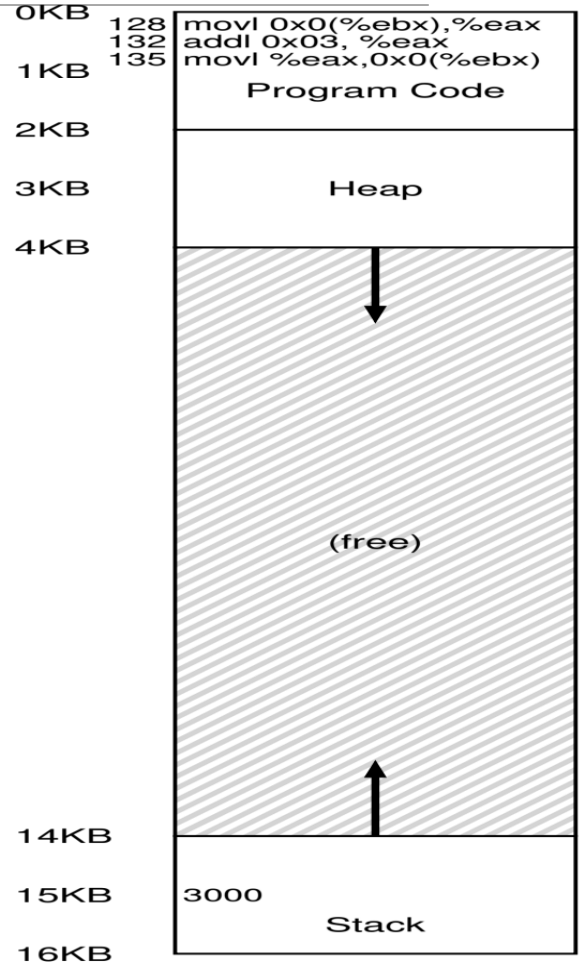
- OS needs memory for its data structures
- For large allocations, OS allocates a page
- For smaller allocations, OS uses various memory allocation algorithms (more later)
  - Cannot use **`libc`** and **`malloc`** in kernel!

Address Translation Mechanism

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

Gets 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

Address Translation

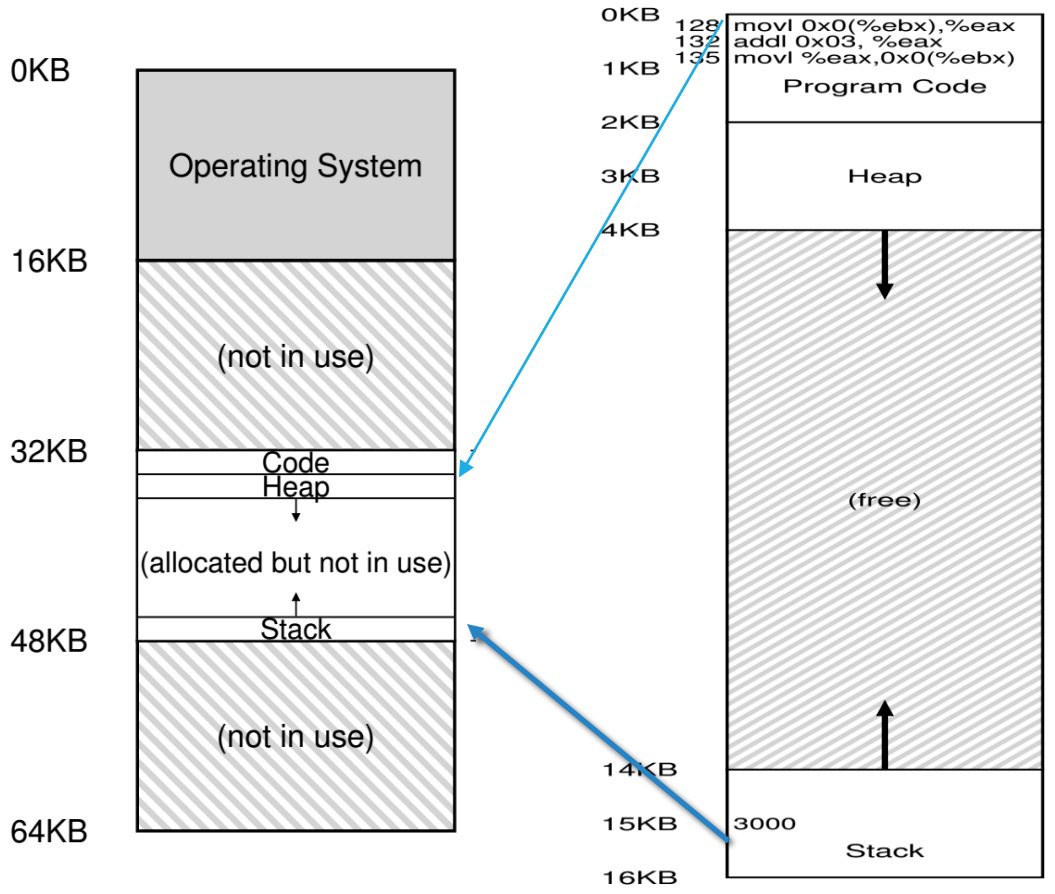
Simplified OS: places entire memory image in one chunk

- Need the following translation from VA to PA

–128 to 32896 (32KB + 128)

–1KB to 33 KB

–20KB? Error!



Who performs address translation

In this simple example, OS tells the hardware the base (starting address) and bound (total size of process) values

- Memory hardware Memory Management Unit (MMU) calculates PA from VA

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

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

Role of Hardware in Translation

- CPU provides privileged mode of execution
- Instruction set has privileged instructions to set translation information (e.g., base, bound)
- Hardware (MMU) uses this information to perform translation on every memory access
- MMU generates faults and traps to OS when access is illegal (e.g., VA is out of bound)

Hardware Requirements	Notes
Privileged mode	*Needed to prevent user-mode processes from executing privileged operations*
Base/bounds registers	*Need pair of registers per CPU to support address translation and bounds checks*
Ability to translate virtual addresses and check if within bounds	*Circuitry to do translations and check limits; in this case, quite simple*
Privileged instruction(s) to update base/bounds	*OS must be able to set these values before letting a user program run*
Privileged instruction(s) to register exception handlers	*OS must be able to tell hardware what code to run if exception occurs*
Ability to raise exceptions	*When processes try to access privileged instructions or out-of-bounds memory*

Role of OS in address Translation

- OS maintains free list of memory
- Allocates space to process during creation (and when asked) and cleans up when done
- Maintains information of where space is allocated to each process (in PCB)
- Sets address translation information (e.g., base &bound) in hardware
- Updates this information upon context switch
- Handles traps due to illegal memory access

OS Requirements	Notes
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*

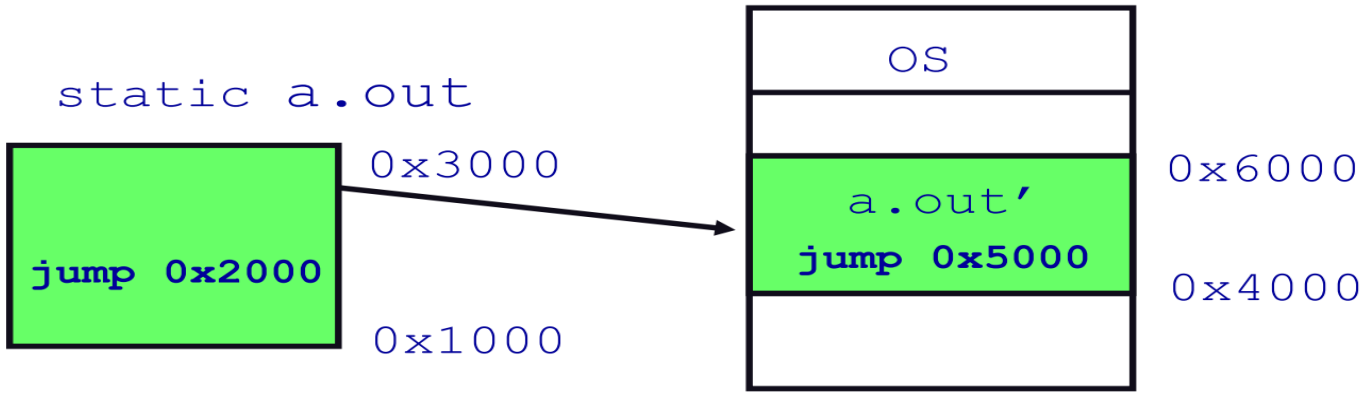
OS at boot time

OS @ boot (kernel mode)	Hardware	(No Program Yet)
initialize trap table		
	remember addresses of...	
	system call handler	
	timer handler	
	illegal mem-access handler	
	illegal instruction handler	
start interrupt timer		
	start timer; interrupt after X ms	
initialize process table		
initialize free list		

OS at Runtime

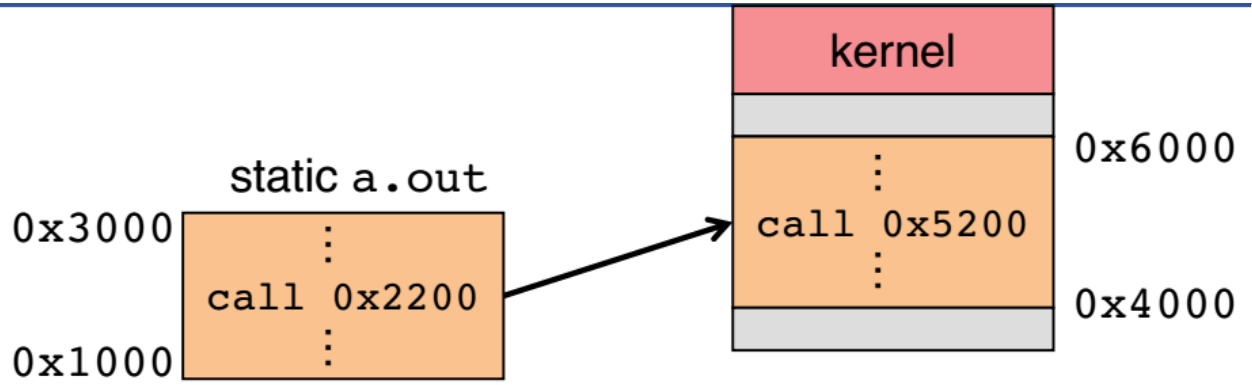
OS @ run (kernel mode)	Hardware	Program (user mode)
To start process A:		
allocate entry in process table		
alloc memory for process		
set base/bound registers		
return-from-trap (into A)		
	restore registers of A	
	move to **user mode**	
	jump to A's (initial) PC	
		Process A runs
		Fetch instruction
	translate virtual address	
	perform fetch	
		Execute instruction
	if explicit load/store:	
	ensure address is legal	
	translate virtual address	
	perform load/store	
		(A runs...)
	Timer interrupt	
	move to **kernel mode**	
	jump to handler	
Handle timer		
decide: stop A, run B		
call `switch()` routine		
save regs(A) to proc-struct(A) (including base/bounds)		
restore regs(B) from proc-struct(B) (including base/bounds)		
return-from-trap (into B)		
	restore registers of B	
	move to **user mode**	
	jump to B's PC	
		Process B runs
		Execute bad load
	Load is out-of-bounds;	
	move to **kernel mode**	
	jump to trap handler	
Handle the trap		
decide to kill process B		
deallocate B's memory		
free B's entry in process table		

Load-time Linking



- **Link as usual, but keep the list of references**
- **Fix up process when actually executed**
  - Determine where process will reside in memory
  - Adjust all references within program (using addition)
- **Problems?**
  - How to enforce protection
  - How to move once in memory (Consider: data pointers)
  - What if no contiguous free region fits program?

Base + Bound Register

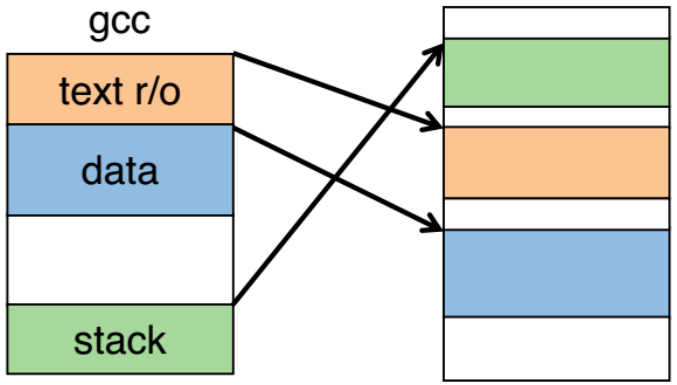


- **Two special privileged registers: base and bound**
- **On each load/store/jump:**
- **How to move process in memory?**
  - Change base register
- **What happens on context switch?**
  - OS must re-load base and bound register

Base + Bound Tradeoffs

- **Advantages**
  - Cheap in terms of hardware: only two registers
  - Cheap in terms of cycles: do add and compare in parallel
  - Examples: Cray-1 used this scheme
- **Disadvantages**
  - Growing a process is expensive or impossible
  - No way to share code or data (E.g., two copies of bochs, both running pintos)
- **One solution: Multiple segments**
  - E.g., separate code, stack, data segments - Possibly multiple data segments

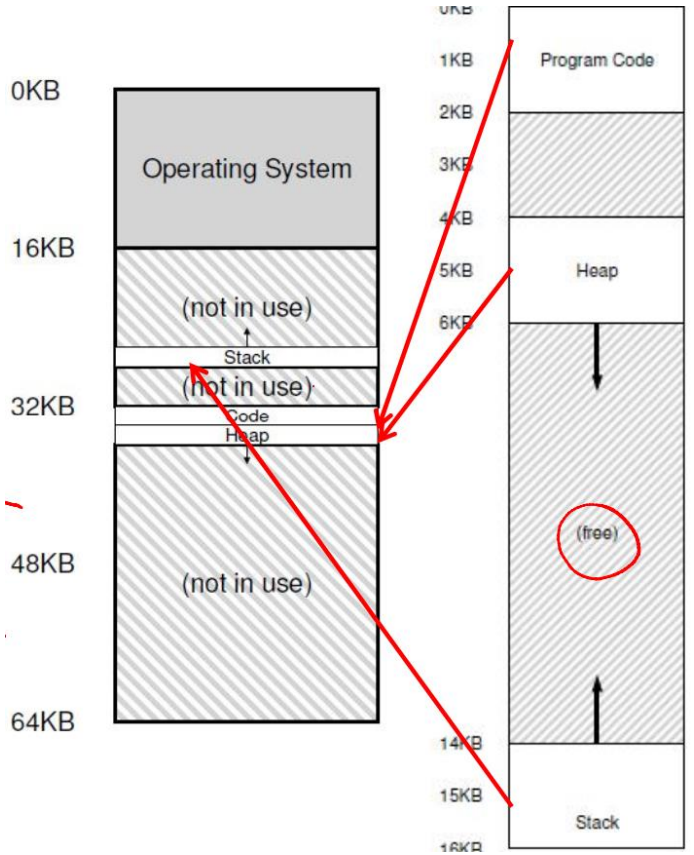
Segmentation



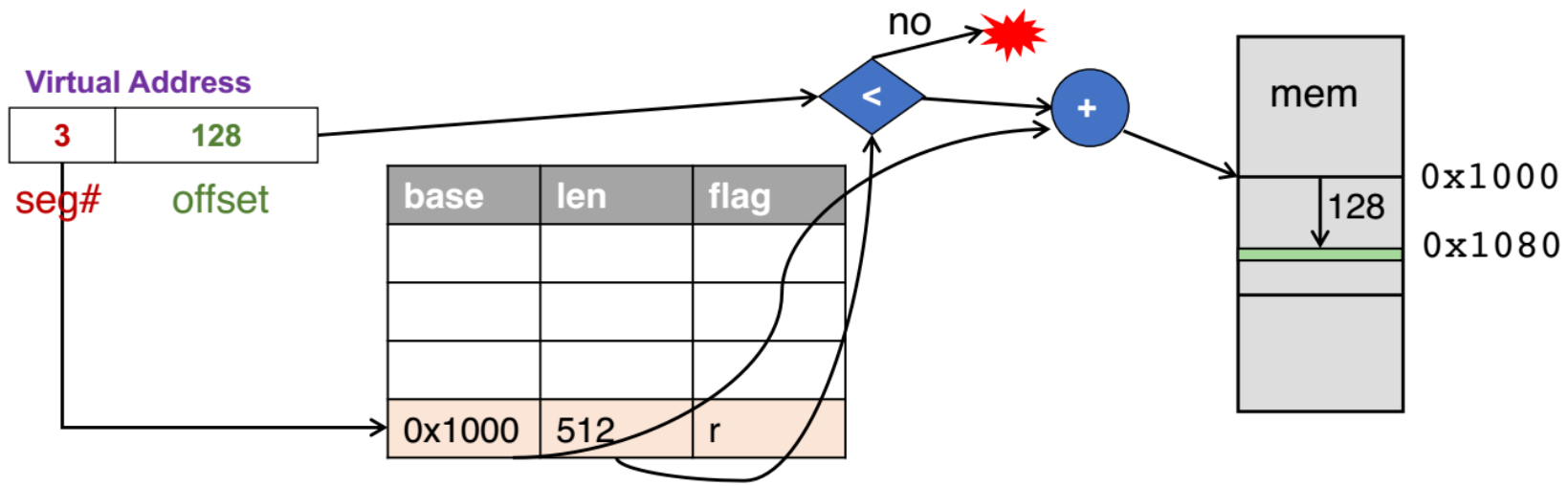
- **Let processes have many base/bound regs**
  - Address space built from many segments
  - Can share/protect memory at segment granularity
- **Must specify segment as part of virtual address**

Segmentation

- Generalized base and bounds
- Each segment of memory image placed separately
- Multiple (base, bound) values stored in MMU
- Good for sparse address spaces
- But variable sized allocation leads to external fragmentation
  - Small holes in memory left between segments



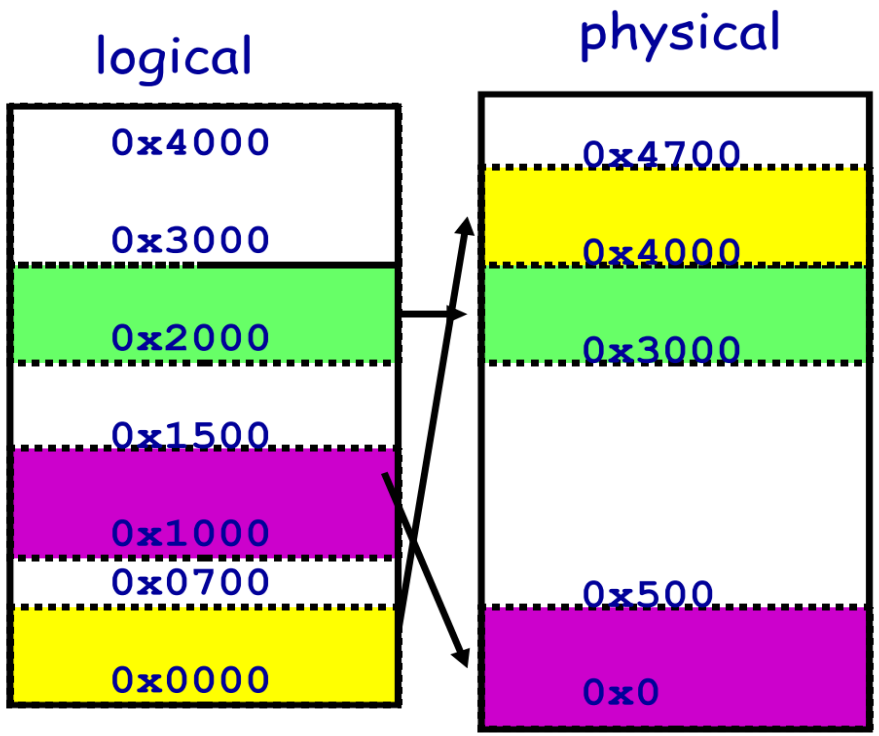
Segmentation Mechanics



- **Each process has a segment table**
- **Each VA indicates a segment and offset:**
  - Top bits of addr select segment, low bits select offset
  - x86 stores segment #s in registers (CS, DS, SS, ES, FS, GS)

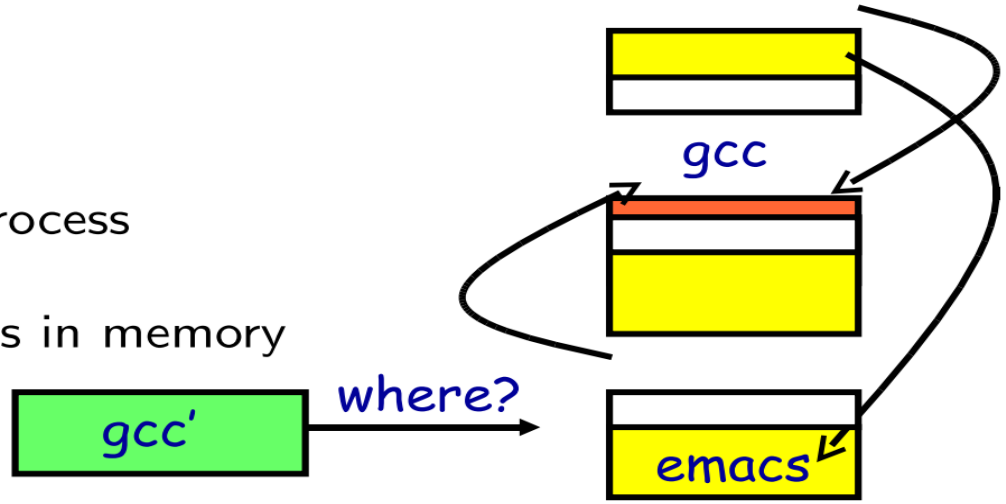
Segmentation Example

Seg	base	bounds	rw
0	0x4000	0x6ff	10
1	0x0000	0x4ff	11
2	0x3000	0xfff	11
3			00



- **2-bit segment number (1st digit), 12 bit offset (last 3)**
  - Where is 0x0240? 0x1108? 0x265c? 0x3002? 0x1600?

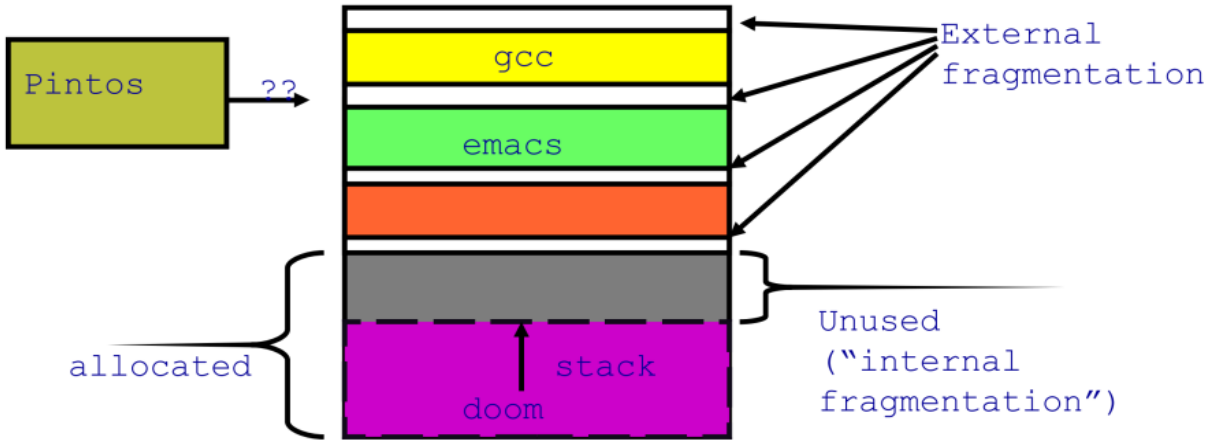
- **Advantages**
  - Multiple segments per process
  - Allows sharing! (how?)
  - Don't need entire process in memory



- **Disadvantages**
  - Requires translation hardware, which could limit performance
  - Segments not completely transparent to program (e.g., default segment faster or uses shorter instruction)
  - *n* byte segment needs *n contiguous* bytes of physical memory
  - Makes *fragmentation* a real problem.

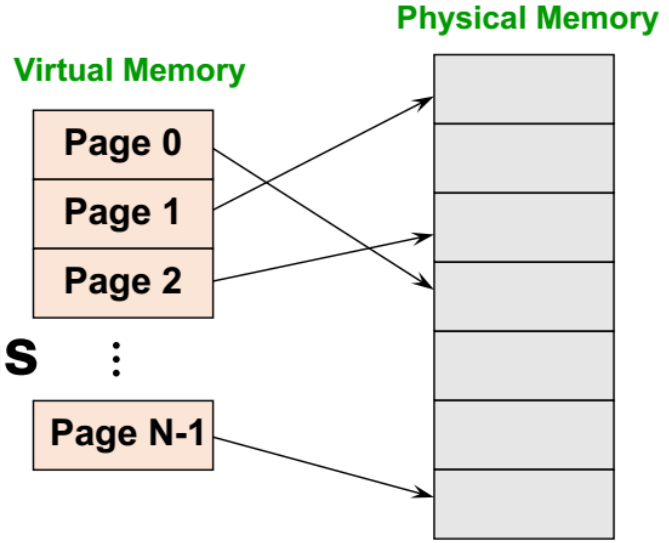
Fragmentation

- **Fragmentation ⇒ Inability to use free memory**
- **Over time:**
  - Variable-sized pieces = many small holes (external fragmentation)
  - Fixed-sized pieces = no external holes, but force internal waste (internal fragmentation)



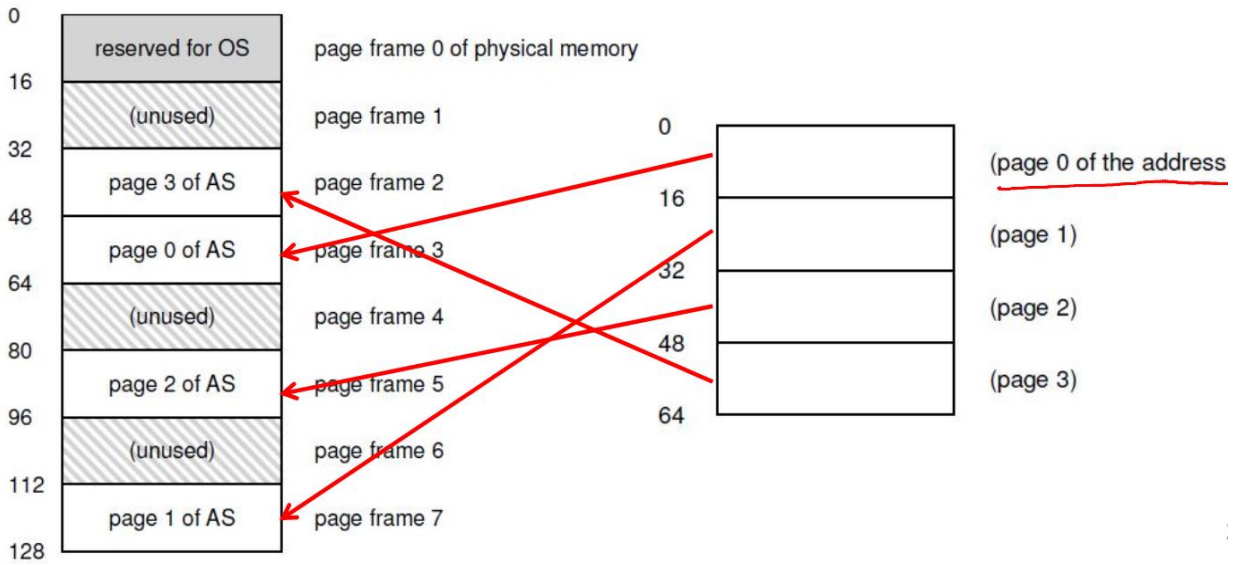
Paging

- **Divide memory up into fixed-size *pages***
  - Eliminates external fragmentation
- **Map virtual pages to physical pages**
  - Each process has separate mapping
- **Allow OS to gain control on certain operations**
  - Read-only pages trap to OS on write
  - Invalid pages trap to OS on read or write
  - OS can change mapping and resume application



Paging

- Allocate memory in fixed size chunks (“pages”)
- Avoids external fragmentation (no small “holes”)
- Has internal fragmentation (partially filled pages)

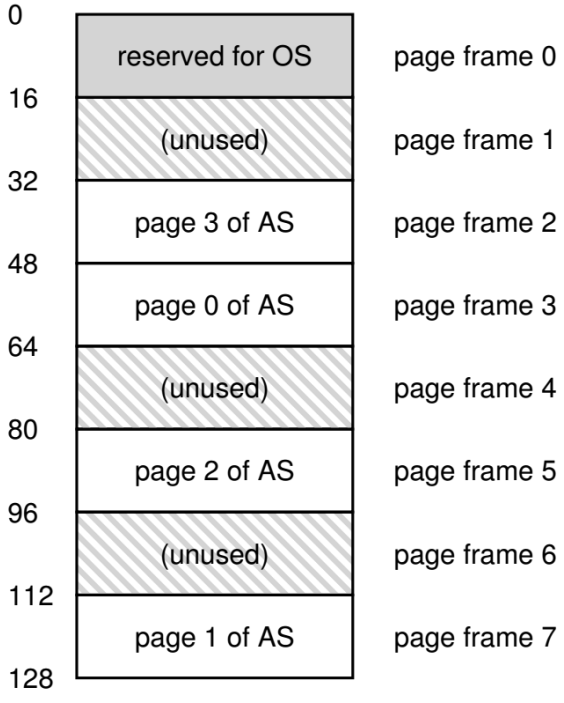


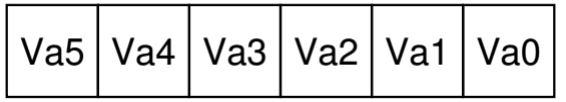
Paging Data structures

- **Pages are fixed size, e.g., 4K**
  - Virtual address has two parts: virtual page number and offset
  - Least significant 12 (log2 4K) bits of address are page offset
  - Most significant bits are *page number*
- **Page tables**
  - Map virtual page number (VPN) to physical page number (PPN)
    - VPN is the index into the table that determines PPN
    - PPN also called page frame number
  - Also includes bits for protection, validity, etc.
  - One page table entry (PTE) per page in virtual address space

Example

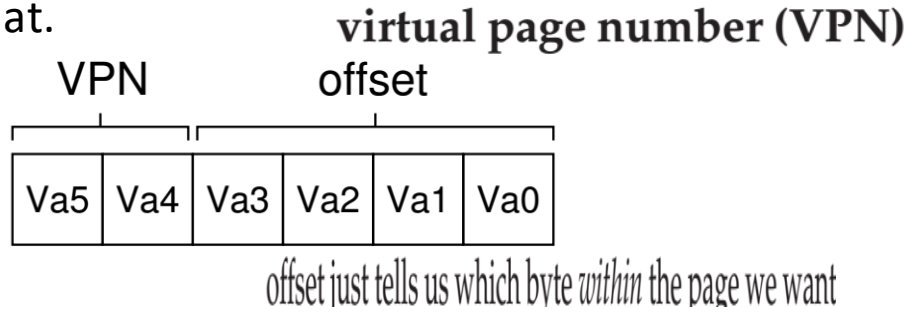
```
movl <virtual address>, %eax
```





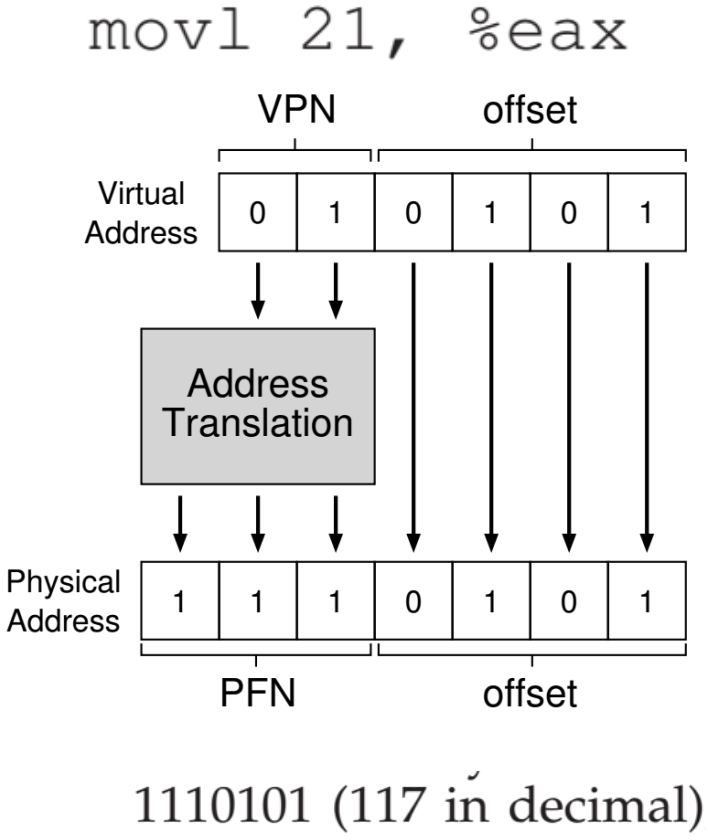
Virtual address space of the process is 64 bytes, we need **6 bits** total for our virtual address (2^6 = 64).

The page size is 16 bytes in a 64-byte address space; thus we need to be able to select 4 pages, and the top 2 bits of the address do just that.

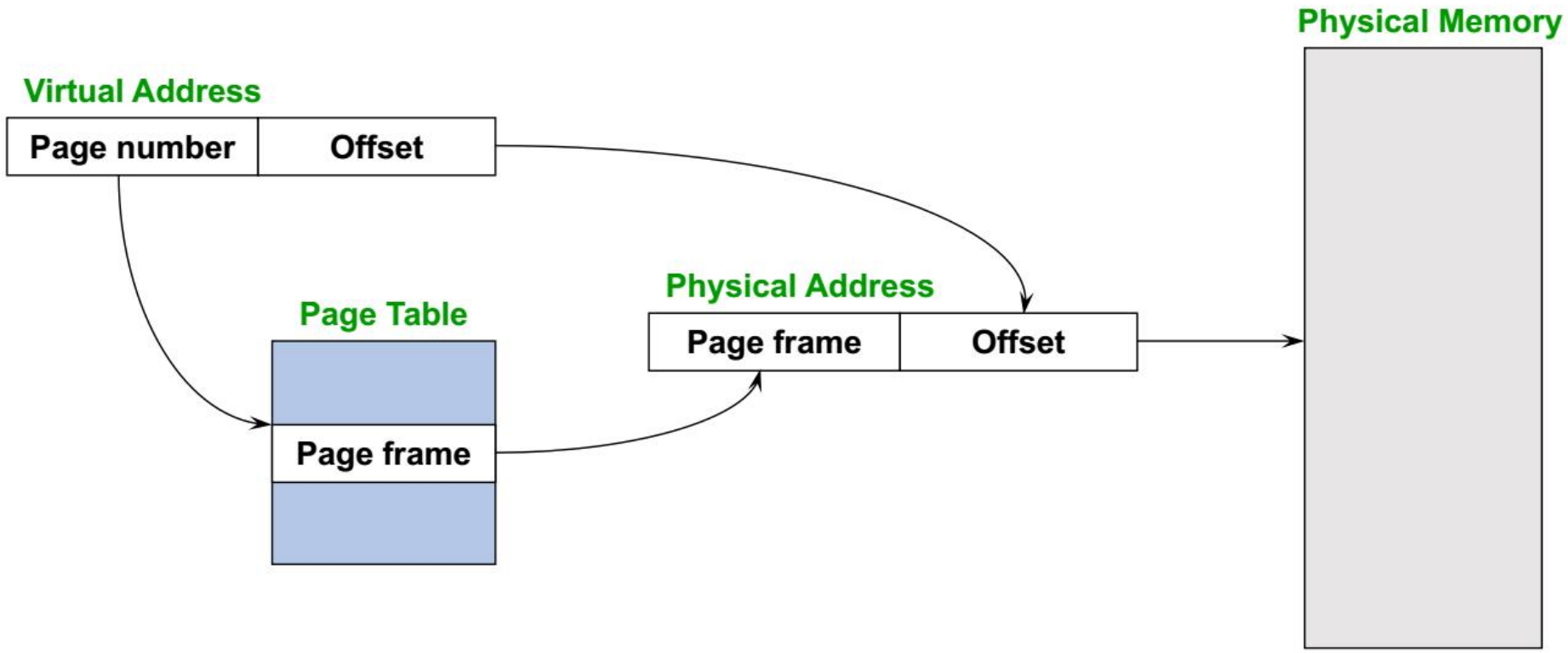


Address Translation in Hardware

- Most significant bits of VA give the VPN
- Page table maps VPN to PFN
- PA is obtained from PFN and offset within a page
- MMU stores (physical)address of start of page table, not all entries.
- "Walks" the page table to get relevant PTE

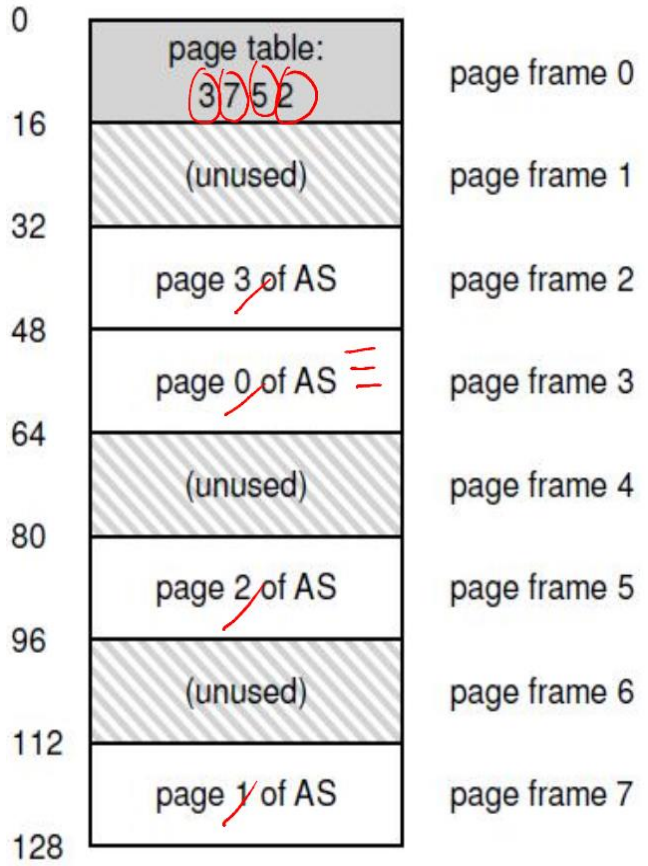


Page Lookups



Page Table

- Per process data structure to help VA-PA translation
- Array stores mappings from virtual page number (VPN) to physical frame number (PFN) – E.g.,VP0 ➡ PF 3, VP 1 ➡ PF 7
- Part of OS memory (in PCB)
- MMU has access to page table and uses it for address Translation
- OS updates page table upon context switch
- Per process data structure to help VA-PA translation



Page Table Entries

Simplest page table: linear page table

- Page table is an array of page table entries, one per virtual page
- VPN (virtual page no.) is index into this array
- Each PTE contains PFN (physical frame number) and few other bits

– Valid bit: is this page used by process?

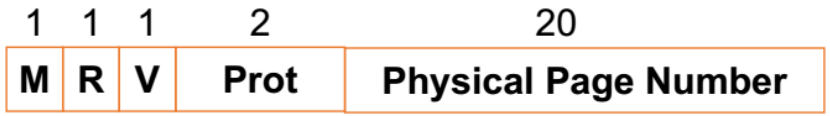
– Protection bits: read/write permissions

– Present bit: is this page in memory? (more later)

– Dirty bit: has this page been modified?

– Accessed bit: has this page been recently accessed?

Page Table Entries (PTEs)



- **Page table entries control mapping**
  - The Modify bit says whether or not the page has been written
    - It is set when a write to the page occurs
  - The Reference bit says whether the page has been accessed
    - It is set when a read or write to the page occurs
  - The Valid bit says whether or not the PTE can be used
    - It is checked each time the virtual address is used
  - The Protection bits say what operations are allowed on page
    - Read, write, execute
  - The Physical page number (PPN) determines physical page

What happens on a memory access ?

```
movl 21, %eax
```

First translate the virtual address (21) into the correct physical address (117).

Before fetching the data from address 117, the system must first fetch the proper page table entry from the process's page table, perform the translation, and then load the data from physical memory. [Hardware must know where the page table is for the currently-running process]

Single **page-table base register** contains the **physical address** of the starting location of the page table.

PTEAddr = Page Table Base Register + (VPN*sizeof(PTE))

Virtual Page Number- used as an index into the array of PTEs pointed to by the page table base register

Now, physical address is known, the hardware can fetch the PTE from memory, extract the PFN, and concatenate it with the offset from the virtual address to form the desired physical address.

```
offset   = VirtualAddress & OFFSET_MASK
PhysAddr = (PFN << SHIFT) | offset
```

The hardware can fetch the desired data from memory and put it into register **`eax`**.

The program has now succeeded at loading a value from memory!

What happens on a memory access ?

```
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT

// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE))

// Fetch the PTE
PTE = AccessMemory(PTEAddr)

// Check if process can access the page
if (PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) == False)
    RaiseException(PROTECTION_FAULT)
else
    // Access is OK: form physical address and fetch it
    offset   = VirtualAddress & OFFSET_MASK
    PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
    Register = AccessMemory(PhysAddr)
```

What happens on a memory access ?

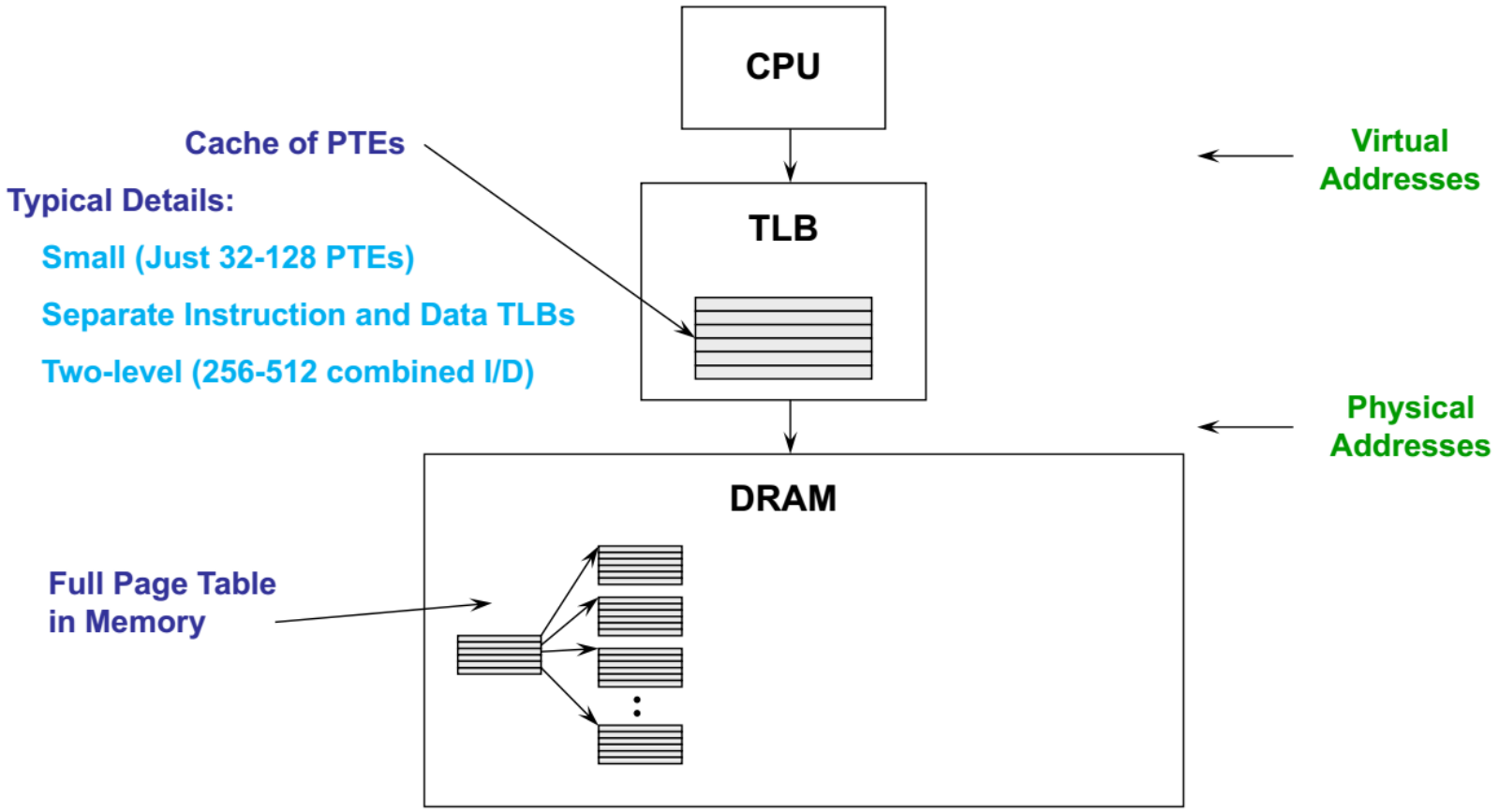
- CPU requests code or data at a virtual address
- MMU must translate VA to PA
  - First, access memory to read page table entry
  - Translate VA to PA
  - Then, access memory to fetch code/data
- Paging adds overhead to memory access
- Solution? A cache for VA-PA mappings

Translation Lookaside Buffer (TLB)

A cache of recent VA-PA mappings

- To translate VA to PA, MMU first looks up TLB
- If TLB hit, PA can be directly used
- If TLB miss, then MMU performs additional memory accesses to "walk" page table
- TLB misses are expensive (multiple memory accesses)
  - Locality of reference helps to have high hit rate
- TLB entries may become invalid on context switch and change of page tables

- **Translation Lookaside Buffers**
  - Translate virtual page #s into PTEs (not physical addrs)
  - Can be done in a single machine cycle
- **TLBs implemented in hardware**
  - Typically 4-way to fully associative cache (all entries looked up in parallel)
  - Cache tags are virtual page numbers
  - Cache values are PTEs (entries from page tables)
  - With PTE + offset, can directly calculate physical address
- **TLBs exploit locality**
  - Processes only use a handful of pages at a time
    - 32-128 entries/pages (128-512K)
    - Only need those pages to be “mapped”
  - Hit rates are therefore very important


CPU
Cache of PTEs
Typical Details:
Small (Just 32-128 PTEs)
Separate Instruction and Data TLBs
Two-level (256-512 combined I/D)
TLB
Virtual
Addresses
Physical
Addresses
DRAM
Full Page Table
in Memory

Paging Advantages

- **Easy to allocate memory**
  - Memory comes from a free list of fixed size chunks
  - Allocating a page is just removing it from the list
  - External fragmentation not a problem
- **Easy to swap out chunks of a program**
  - All chunks are the same size
  - Use valid bit to detect references to swapped pages
  - Pages are a convenient multiple of the disk block size

Limitations

- **Can still have internal fragmentation**
  - Process may not use memory in multiples of a page
- **Memory reference overhead**
  - 2 or more references per address lookup (page table, then memory)
  - Solution – use a hardware cache of lookups (more later)
- **Memory required to hold page table can be significant**
  - Need one PTE per page
  - 32 bit address space w/ 4KB pages = 2^{20} PTEs
  - 4 bytes/PTE = 4MB/page table
  - 25 processes = 100MB just for page tables!
  - Solution – page the page tables (more later)

How are page tables stored in memory

- What is typical size of page table?
  - 32 bit VA, 4 KB pages, so 2^32 / 2^12 = 2^20 entries
  - If each PTE is 4 bytes, then page table is 4MB
  - One such page table per process!
- How to reduce the size of page tables?
  - Larger pages, so fewer entries
- How does OS allocate memory for such large tables?
  - Page table is itself split into smaller chunks!

Managing Page tables

- **Size of the page table for a 32-bit address space w/ 4K pages is 4MB**
  - This is far far too much overhead for each process
- **How can we reduce this overhead?**
  - Observation: Only need to map the portion of the address space actually being used (tiny fraction of entire addr space)
- **How do we only map what is being used?**
  - Can dynamically extend page table…
  - Does not work if addr space is sparse (internal fragmentation)
- **Use another level of indirection: two-level page tables**