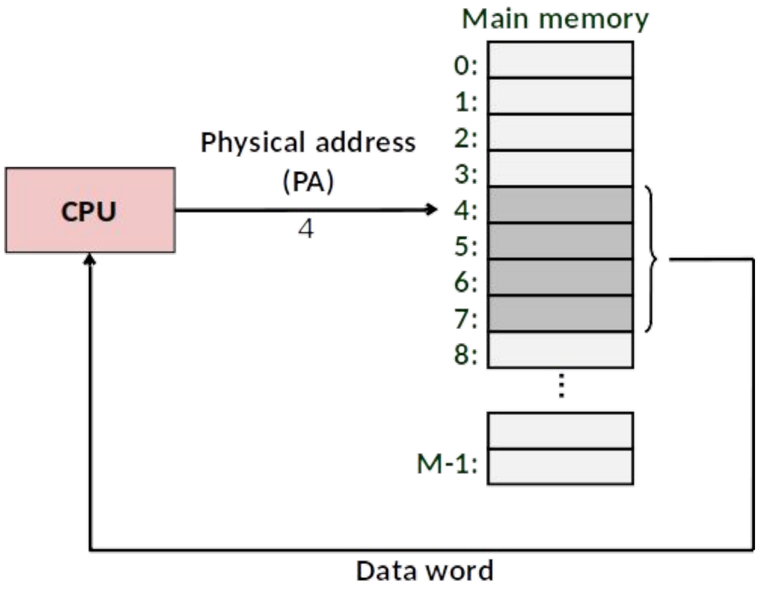
09. Main Memory

CS 4352 Operating Systems

Adopted from slides of Drs. Silberschatz, Huang, & Bryant

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
  - Main memory and registers are only storage CPU can access directly

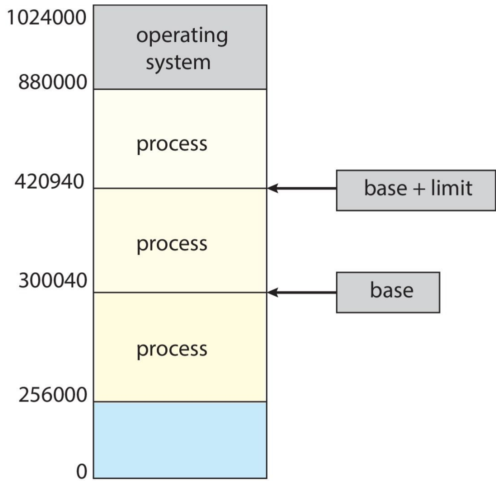


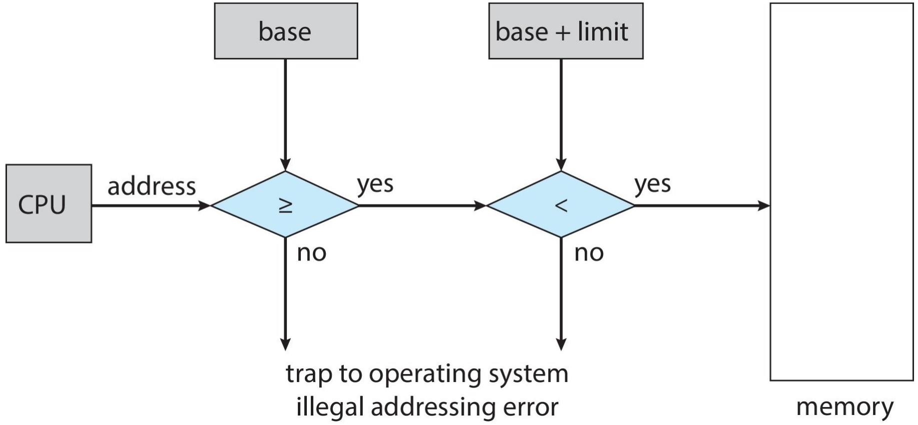
Multiprogramming Issues

- We want multiple processes in memory at once
    - Process A may access any physical address
- Protection of memory required to ensure correct operation
    - Need to ensure that a process can access only access those addresses belonging to it
    - Otherwise …

How About?

- We may try to provide this protection by using a pair of **base** and **limit registers** to define the address space of a process
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user
    - The instructions to loading the base and limit registers are privileged



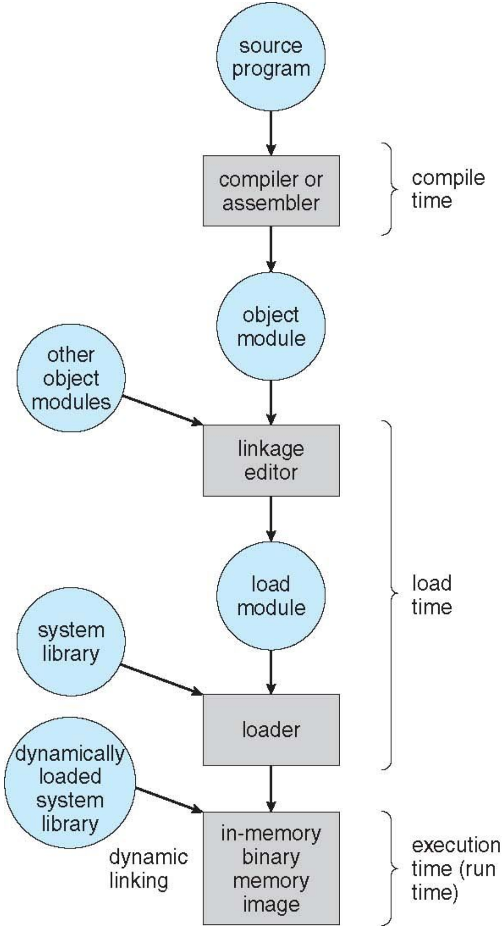


Problems

- What happens if a process needs to expand?
- What if a process needs more memory than available?
- When does a process have to know it will run at 0x300040?
- What if a process isn’t using its memory?

Address Binding

- Address binding of instructions and data to memory addresses can happen at three different stages
  - Compile time: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes
  - Load time: Must generate relocatable code if memory location is not known at compile time
  - Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)



Dynamic Loading

- The entire program does need to be in memory to execute
  - Routine is not loaded until it is called
  - Better memory-space utilization; unused routine is never loaded
  - All routines kept on disk in relocatable load format
  - Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading

Dynamic Linking

- Static linking – system libraries and program code combined by the linker into the binary program image (e.g., gcc -static)
- Dynamic linking – linking postponed until execution time
  - Small piece of code, stub, used to locate the appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes the routine
  - Operating system checks if routine is in processes' memory address
    - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
  - Also known as **shared libraries**

Virtual Memory

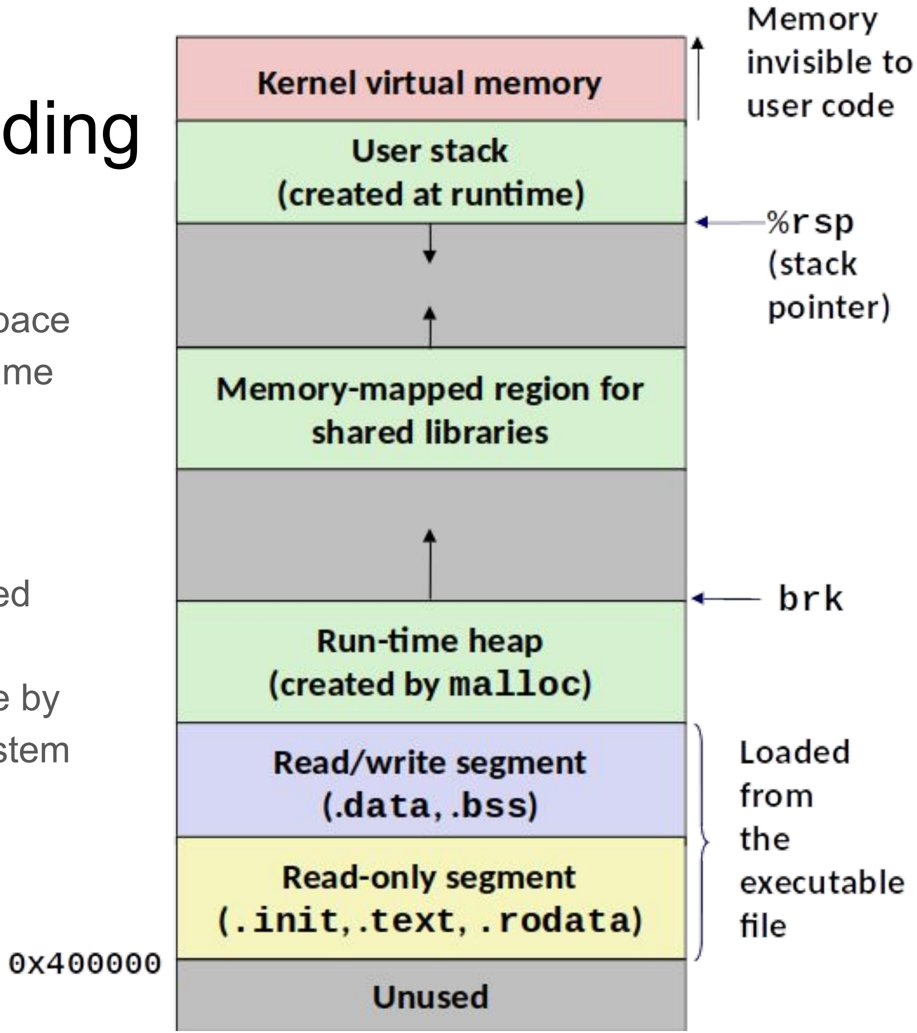
- The abstraction that the OS provides for managing memory
    - VM enables a program to execute with less physical memory than it “needs”
    - Many programs do not need all of their code and data at once (or ever) – no need to allocate memory for it
    - OS will adjust memory allocation to a process based upon its behavior
    - VM requires hardware support and OS management algorithms to pull it off
- Goals
    - Give each process its own virtual address space
        - Application doesn’t see physical memory addresses
    - Enforce protection
        - Prevent one process from messing with another’s memory
    - Allow programs to see more memory than exists

Virtual vs. Physical Address Space

- The concept of a **virtual address space** that is bound to a separate **physical address space** is central to proper memory management
  - Virtual address – generated by the CPU
    - Virtual address space is the set of all virtual addresses generated by a program
  - Physical address – address seen by the memory unit
    - Physical address space is the set of all physical addresses generated by a program
- Virtual to physical address translation is something user process doesn’t know

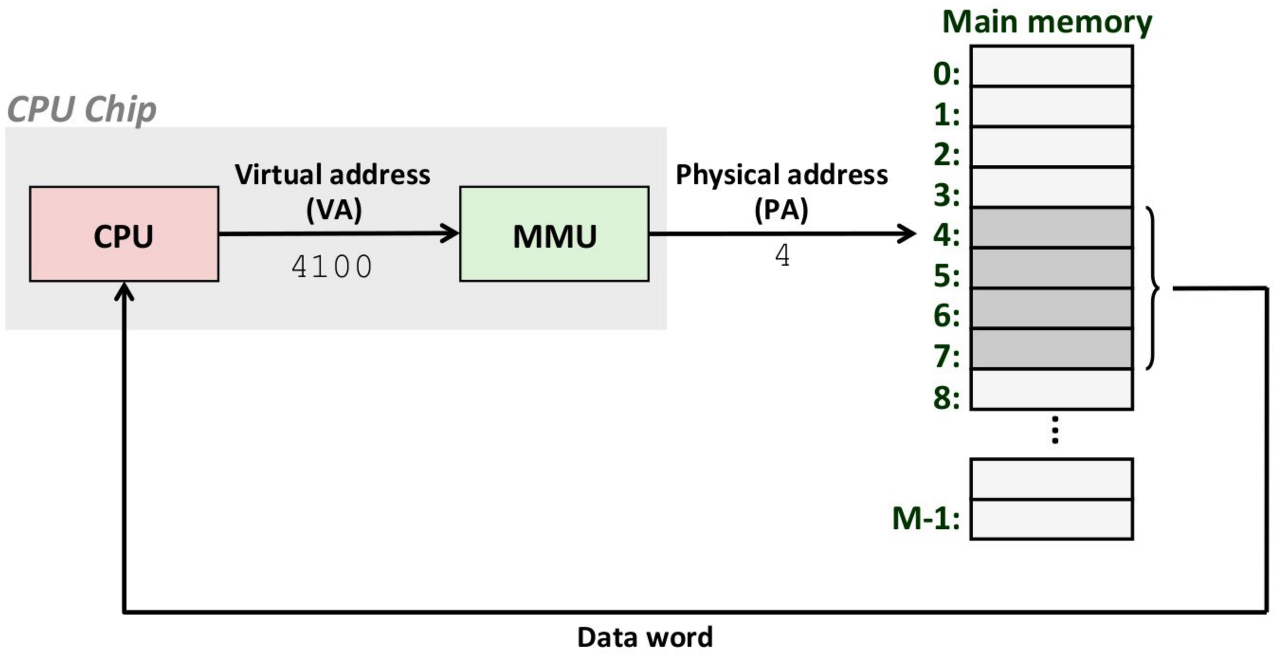
Simplifying Linking and Loading

- Linking
  - Each program has similar virtual address space
  - Code, data, heap can always start at the same address
- Loading (we will learn paging later)
  - execve() allocates pages for text and data sections & creates page table entries marked as invalid
  - The text and data sections are copied, page by page, on demand by the virtual memory system



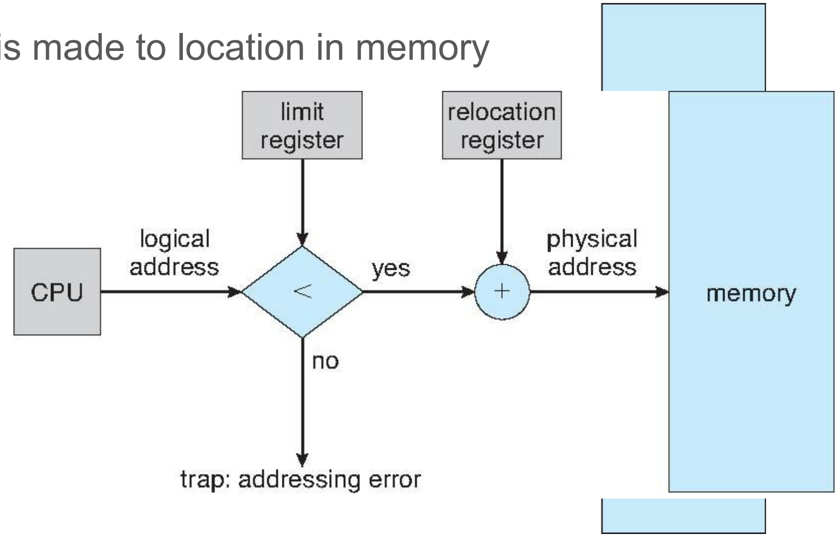
Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address



Generalization of Base+Limit Scheme

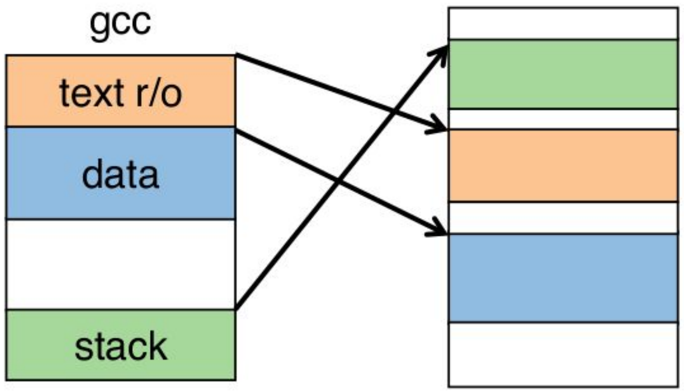
- The base register now called relocation register
  - The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with virtual addresses; it never sees the real physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Virtual address bound to physical addresses
  - Limit register contains range of virtual addresses – each virtual address must be less than the limit register
  - MMU maps virtual address dynamically



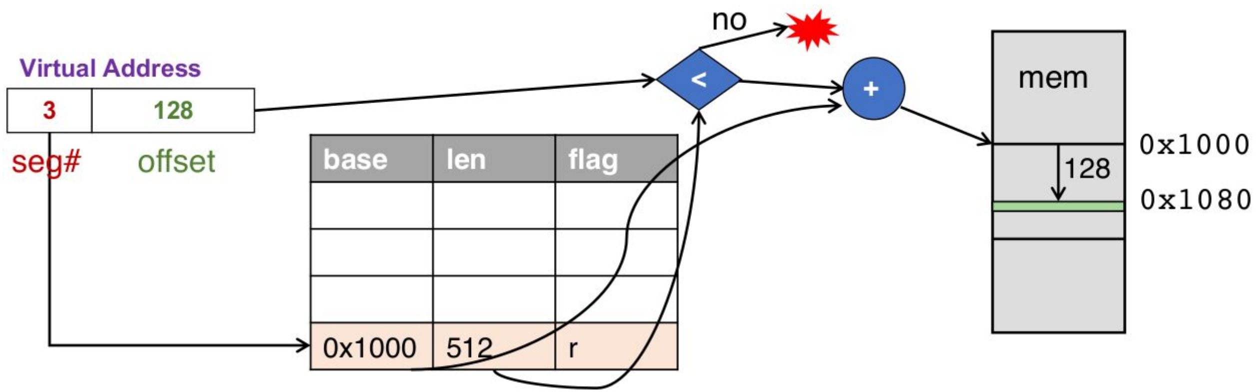
Base+Limit Trade-offs

- 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
- One solution: Multiple segments
  - E.g., separate code, stack, data segments - possibly multiple data segments

Segmentation

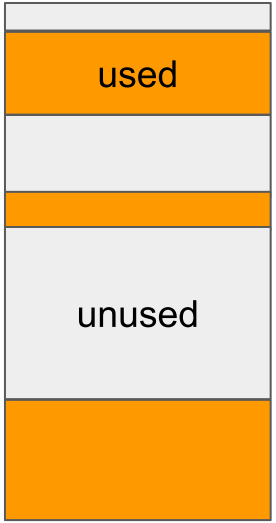


- Let processes have many bases and limits
  - Address space built from many segments
  - Can share/protect memory at segment granularity
- Each process has a segment table
- Each virtual address indicates a segment and offset:
  - Top bits of address select segment, low bits select offset
  - x86 stores segment #s in registers (CS, DS, SS, ES, FS, GS)



Allocation Problem

- When a process arrives, it needs pieces of contiguous memory to accommodate its segments
  - Blocks of available memory; holes of various size are scattered throughout memory
  - How to satisfy a request of size n from a list of free holes?
    - First-fit: Allocate the first hole that is big enough
    - Best-fit: Allocate the smallest hole that is big enough; must search entire list, unless ordered by size
      - Produces the smallest leftover hole
    - Worst-fit: Allocate the largest hole; must also search entire list
      - Produces the largest leftover hole



Fragmentation

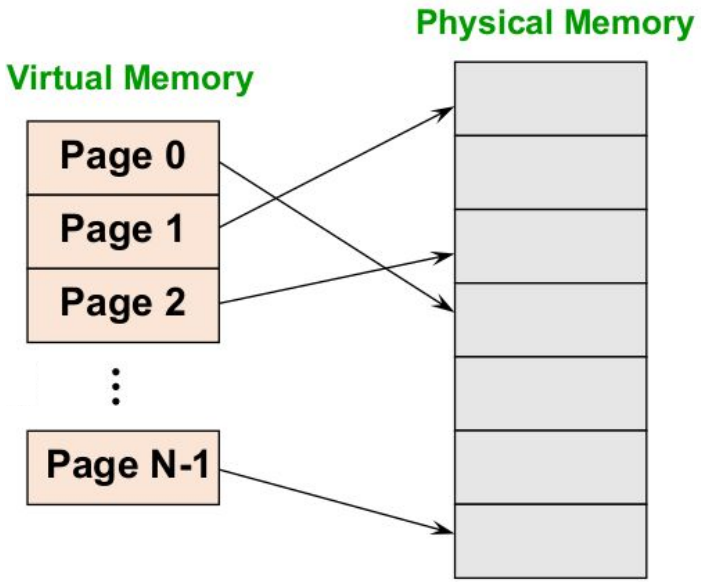
- External fragmentation – total memory space exists to satisfy a request, but it is not contiguous
  - Reduce external fragmentation by compaction
    - Shuffle memory contents to place all free memory together in one large block
    - Compaction is possible only if relocation is dynamic, and is done at execution time
- Internal fragmentation – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given N blocks allocated, 0.5 N blocks lost to fragmentation
  - 1/3 may be unusable -> 50-percent rule

Wakeup!

Things are becoming important!!!!!

Paging

- Divide physical memory up into fixed-size **page frames**
  - Size is power of 2, normally 4 KB (2^12)
- Divide logical memory into blocks of same size called pages
- Map virtual pages to physical page frames
  - Each process has separate mapping



Paging Data Structures

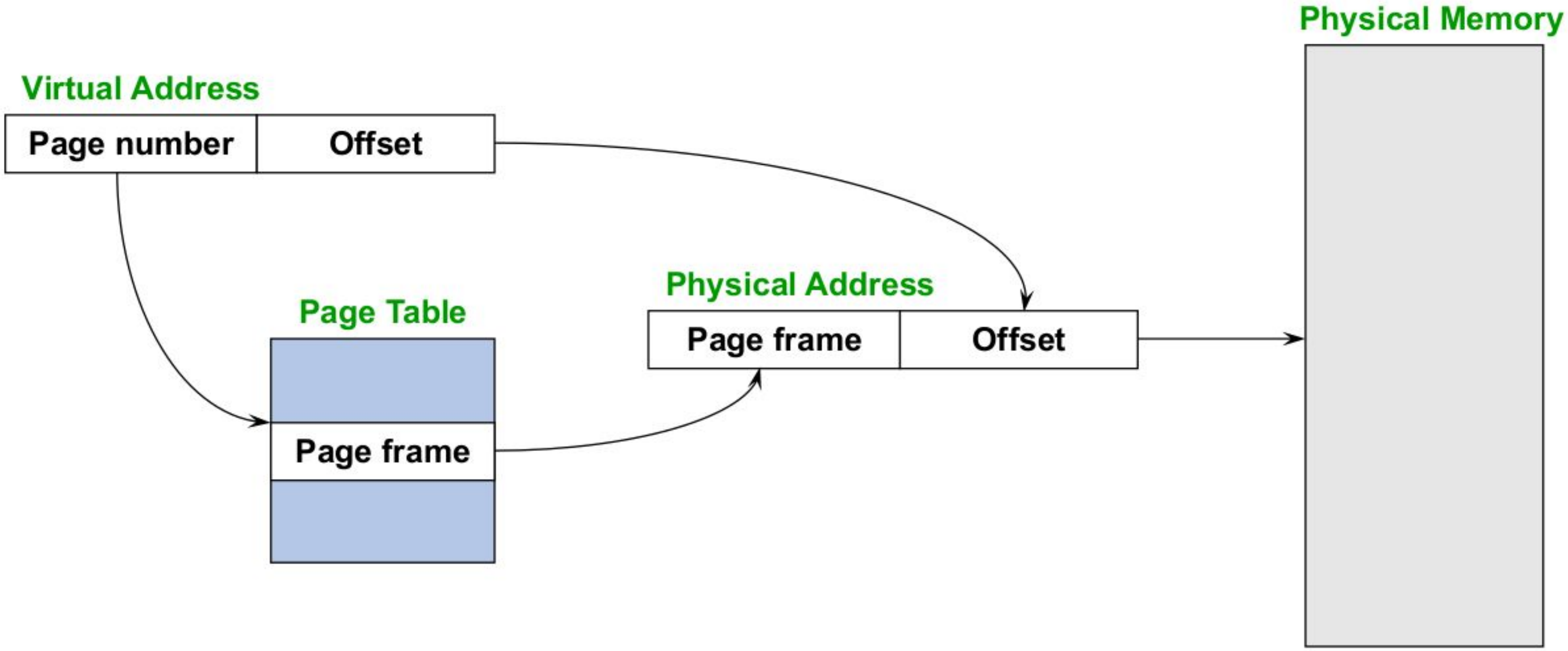
- Pages are fixed size, e.g., 4KB
  - Virtual address has two parts: virtual **page number** and **page offset**
  - Least significant 12 (log2 4K) bits of address are page offset
  - Most significant bits are page number
- Page tables
  - Map **virtual page number** (VPN) to **page frame number** (PFN)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

Page Table Entries (PTEs)

- Page table entries control mapping
- An example on a 32-bit machine
    - 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

1	1	1	2	20
M	**R**	**V**	**Prot**	**Physical Page Number**

Page Lookups



Paging Example

- Let us assume pages are 4KB on a 32-bit system
  - VPN is 20 bits (2^20 VPNs), page offset is 12 bits
- Virtual address is 0xFEDA7468
  - Virtual page is 0xFEDA7, page offset is 0x468
- Page table entry 0xFEDA7 contains 0x2
  - Page frame number is 0x2
  - The 0xFEDA7 th virtual page is at address 0x2000 (2nd physical page frame)
- Physical address = 0x2000 + 0x468 = 0x2468

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

Paging Limitations

- Can still have internal fragmentation
  - Process may not use all the memory in a page frame
- 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 with 4KB pages = 2^20 PTEs
    - 4 bytes/PTE = 4MB/page table
    - 25 processes = 100MB just for page tables!
  - How about 64-bit address space?
  - Solution – page the page tables (more later)

Next Lecture

Continue studying paging!