Memory Management

Today

- Basic memory management
- Swapping
- Kernel memory allocation

Next Time

- Virtual memory

Memory management

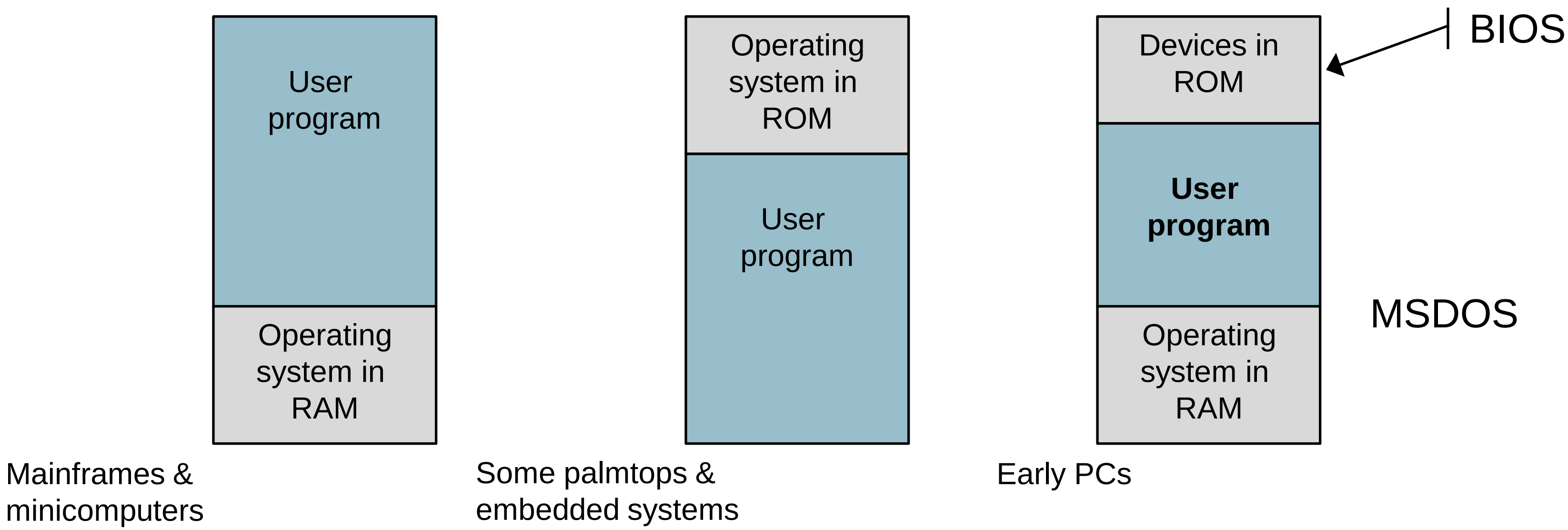
- Ideal memory for a programmer
  - Large
  - Fast
  - Non volatile
  - Cheap
- Nothing like that → memory hierarchy
  - Small amount of fast, expensive memory – cache
  - Some medium-speed, medium price main memory
  - Gigabytes of slow, cheap disk storage
- Memory manager handles the memory hierarchy
  - Allocates scarce resource given competing requests to maximize memory utilization and system throughput
  - Offers a convenient abstraction for programming
  - Provides isolation between processes

Virtual memory

- A basic abstraction commonly supported by all OS for desktops and servers today
  – Efficient use of physical memory – processes can execute without needing to have all their address space in memory
  – Program flexibility – processes can run in machines with less physical memory than they need
  – Protection – virtual memory isolates the address spaces of processes from each other
- Of course, virtual memory needs hardware and OS support
  – Today and the next few lectures
- But first let’s take a history detour …

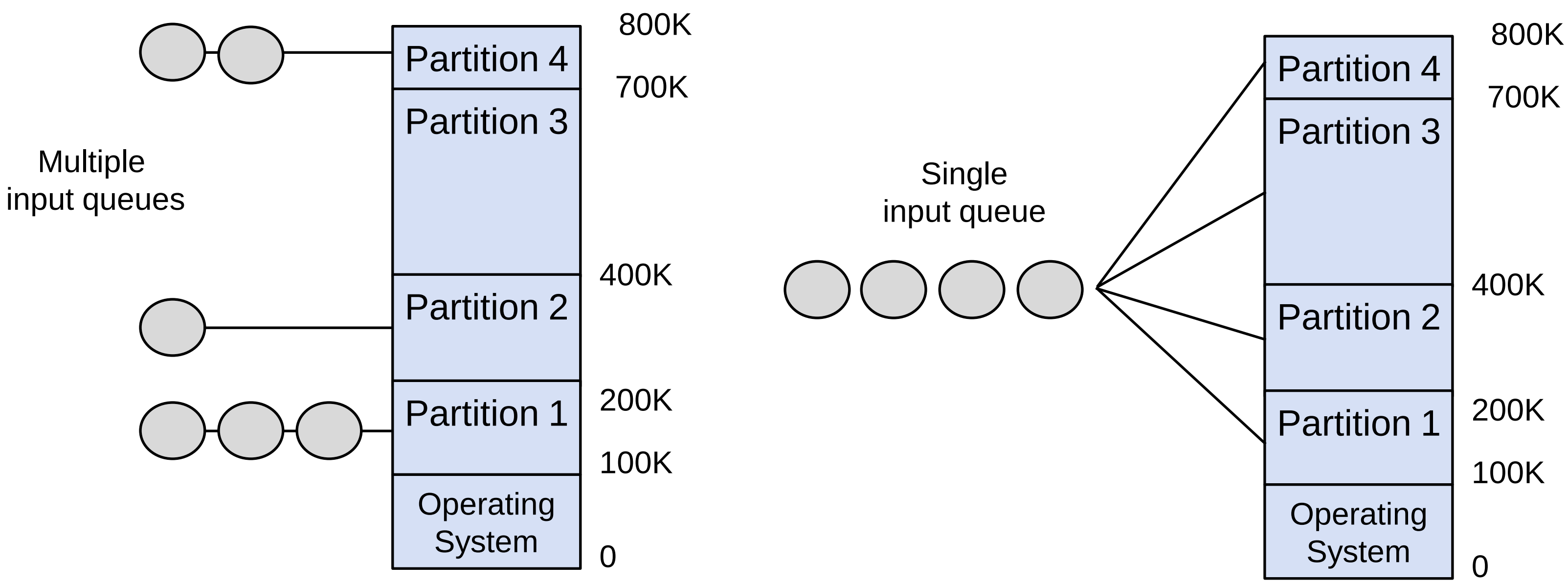
Basic memory management

- Simplest memory abstraction – no abstraction at all
  - Early mainframes (before ‘60), minicomputers (before ‘70) and PCs (before ‘80)
  - `MOV REG1, 1000` moves content of physical memory 1000 to register 1
  - Logically, only one program running at a time *Why?*
  - Still here, some alternatives for organizing memory



Multiprogramming

- With a bit of hardware – Multiprogramming – while one process waits for I/O, another one can use the CPU
- Multiprogramming with fixed partitions
  - Split memory in *n* parts (possible != sizes)
  - Single or separate input queues for each partition
  - ~IBM OS/360 – MFT: Multiprogramming with Fixed number of Tasks

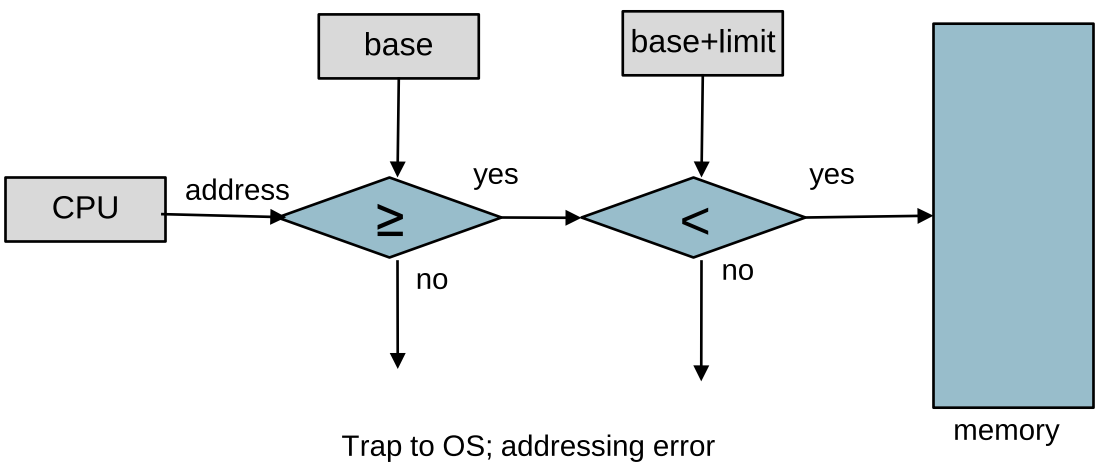


Two problems w/ multiprogramming

- Protection and relocation
  - Keep a process out of other processes' partitions
    - IBM 360 - modify instructions on the fly
      - Split memory into 2KB blocks,
      - Add key/code combination (4 bit)
      - Key was kept in a register (PSW, program status word) with other condition code bits
      - OS trapped any process trying to access memory with protection != the PSW key
  - Don't know where a program will be loaded in memory
    - Address locations of variables & code routines
    - IBM 360 – modify program at loading time (static relocation)
- Better, a new abstraction: Address space
  - The set of addresses a process can use to address memory
  - Each process has its own address space

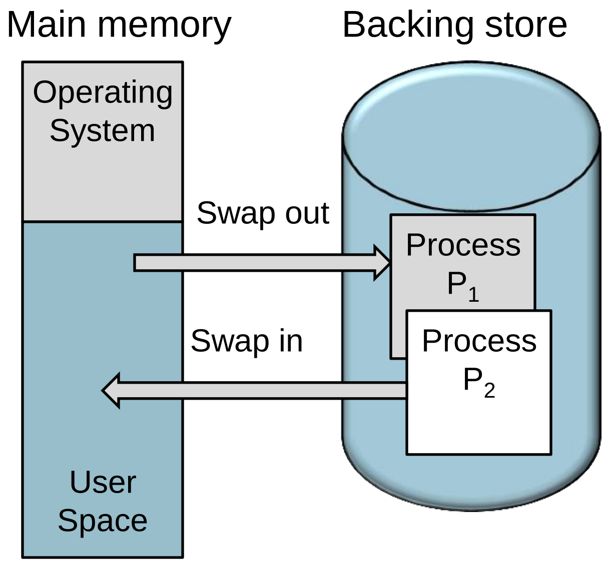
Two problems w/ multiprogramming

- Use base and limit values (CDC 6600 & Intel 8088)
  - Address locations + base value → physical address
  - Ideally, the base and limit registers can only be modified by the OS
  - A disadvantage – Comparisons can be done fast but additions can be expensive



Swapping

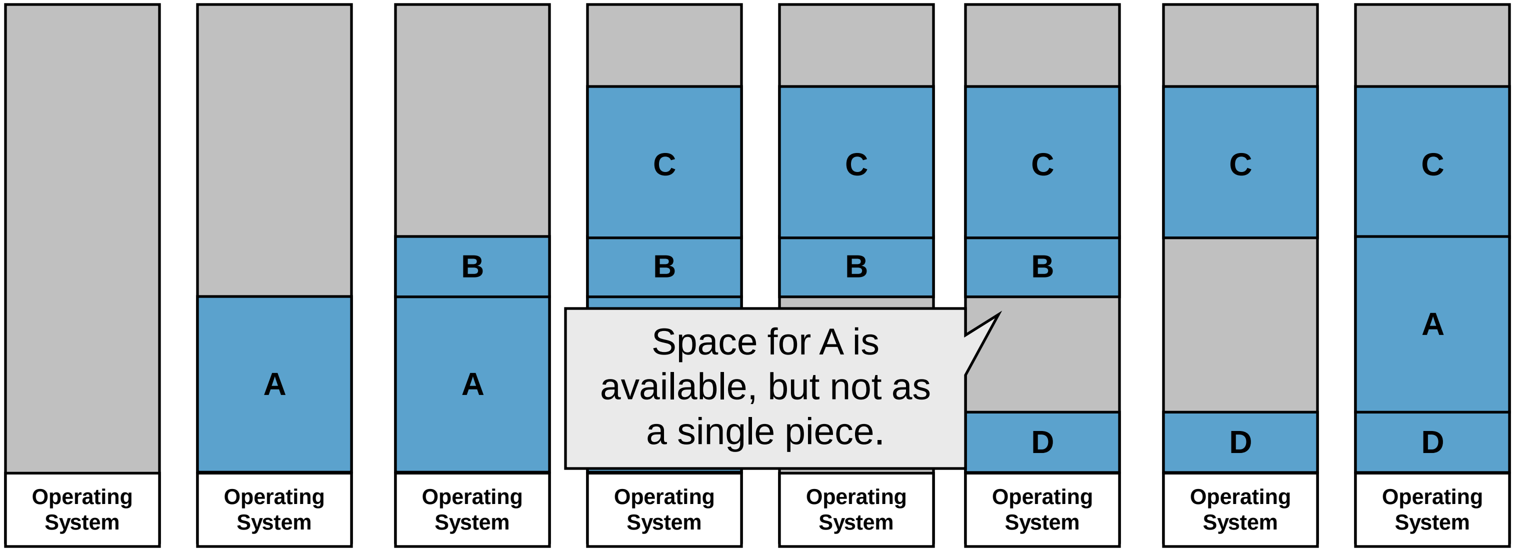
- If physical memory is enough to hold all processes, then we are mostly done
- Not enough memory for all processes?
  – Swapping
    - Simplest
    - Bring each process entirely
    - Move another one to disk
    - Compatible Time Sharing System (CTSS) – a uniprogrammed swapping system



  – Virtual memory (your other option)
    - Allow processes to be only partially in main memory

Swapping

- How is different from MFT?
  - Much more flexible
    - Size & number of partitions changes dynamically
  - Higher memory utilization, but harder memory management
- Swapping in/out creates multiple holes
  - Fragmentation …

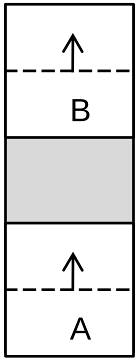


Fragmentation

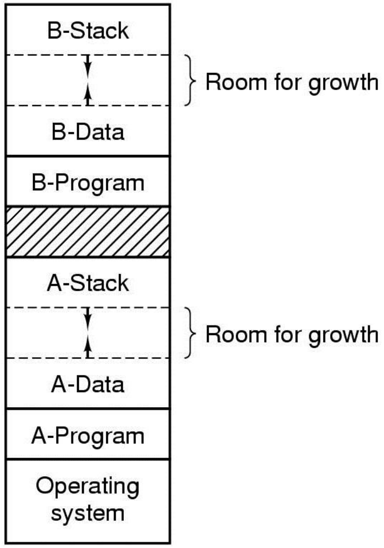
- External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous
- Reduce external fragmentation by compaction
  - Shuffle contents to group free memory as one block
  - Possible only if relocation is dynamic; done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
- Too expensive (1GB machine that can copy at 4B/20nsec will take 5 sec to compact memory!)

How much memory to allocate?

- If process' memory doesn't grow – easy
- In real world, memory needs change dynamically:
  - Swapping to make space?
  - Allocate more space to start with
    - Internal Fragmentation – leftover memory is internal to a partition
  - Remember what you used when swapping



- More than one growing area per processes
  - Stack & data segment
  - If need more, same as before



Memory management

- With dynamically allocated memory
  - OS must keep track of allocated/free memory
  - Two general approaches - bit maps and linked lists
- Bit maps
  - Divide memory into allocation units, track usage with a bitmap
  - Design issues - Size of allocation unit
    - The smaller the size, the larger the bitmap
    - The larger the size, the bigger the waste
  - Simple, but slow – find a big enough chunk?
- Linked list of allocated or free spaces
  - List ordered by address
  - Double link will make your life easier
    - Updating when a process is swapped out or terminates

Picking a place – different algorithms

- If list of processes & holes is ordered by addresses, different ways to get memory for a new processes …
  - First fit – simple and fast
  - Next fit - ~ First fit but start where it left off
    - Slightly worst performance than First fit
  - Best fit – try to waste the least but …
    - More wasted in tiny holes!
  - Worst fit – try to “waste” the most (easier to reuse)
    - Not too good either
  - Speeding things up
    - Two lists (free and allocated) – slows down de-allocation
    - Order the hole list – first fit ~ best fit
    - Use the same holes to keep the list
    - Quick fit – list of commonly used hole sizes - allocation is quick, merging is expensive

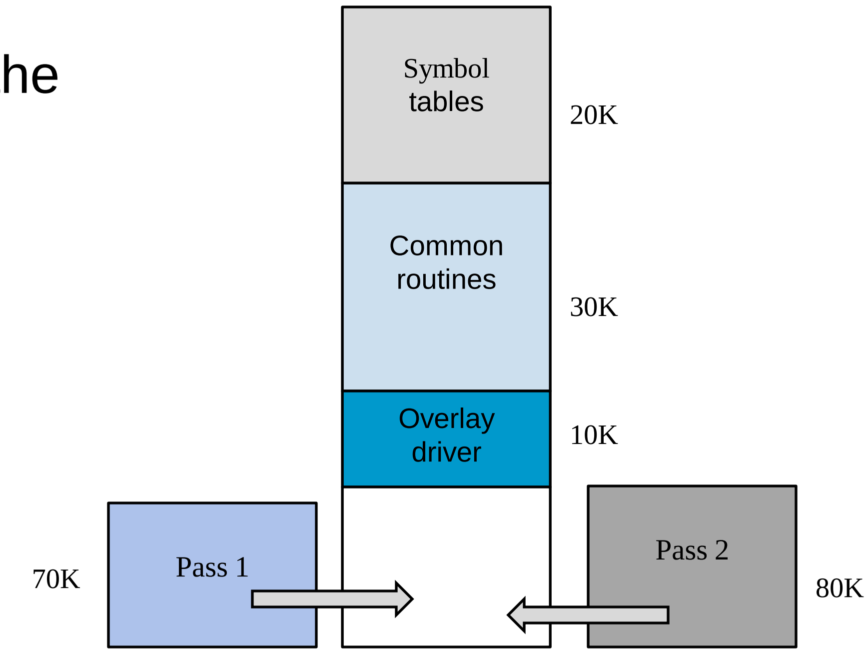
Virtual memory

- *Handling processes >> than allocated memory*
- Keep in memory only what's needed
  - Full address space does not need to be resident in memory
  - OS uses main memory as a cache
- Overlay approach
  - Implemented by user
  - Easy on the OS, hard on the programmer

Overlay for a two-pass assembler:

Pass 1	70KB
Pass 2	80KB
Symbol Table	20KB
Common Routines	30KB
Total	200KB

Two overlays: 120 + 130KB

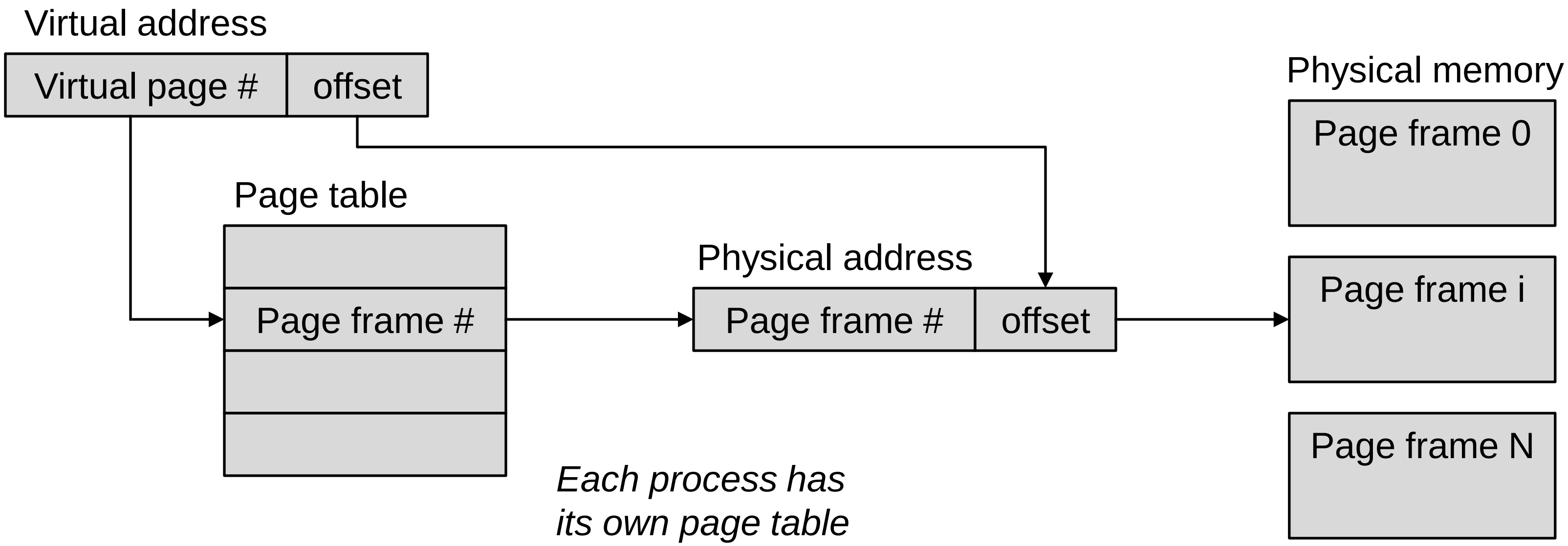


Virtual memory – paging

- Paging – hide the complexity, let the OS do the job
  - Virtual address space split into pages, each a contiguous range of addresses;
  - Physical memory split into page frames
  - Pages are mapped onto frames
    - Doing the translation – OS + MMU
    - Not all have to be in at once
    - If page is in memory, system does the mapping, if not the OS is told to get the missing page and re-execute the failed instruction
- Good for everyone
  - Developers – memory seems a contiguous address space with size independent of hardware
  - Mem manager can efficiently use mem with minimal internal (small units) & no external fragmentation (fixed size units)
  - Protection since processes can't access each other's memory

Address translation with paging

- Virtual to physical address
  - Two parts – virtual page number and offset
  - Virtual page number – index into a page table
  - Page table maps virtual pages to page frames
    - Managed by the OS
    - One entry per page in virtual address space
  - Physical address – page number and offset



Pages, page frames and tables

A simple example with

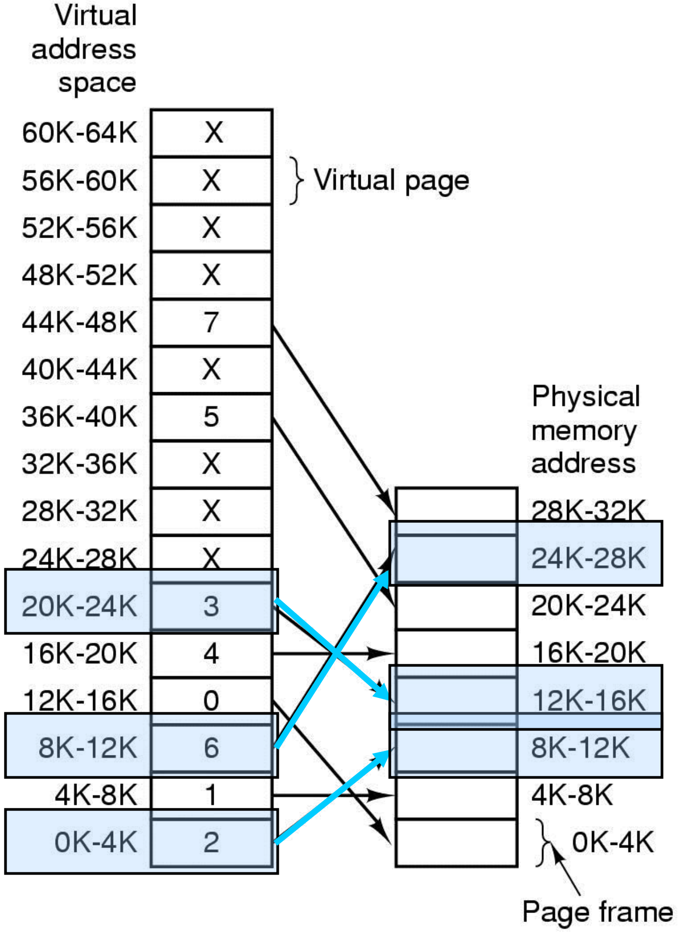
- 64KB virtual address space
- 4KB pages
- 32KB physical address space
- 16 pages and 8 page frames

Try to access :

- **MOV REG, 0**
  Virtual address 0
  Page frame 2
  Physical address 8192

- **MOV REG, 8192**
  Virtual address 8192
  Page frame 6
  Physical address 24576

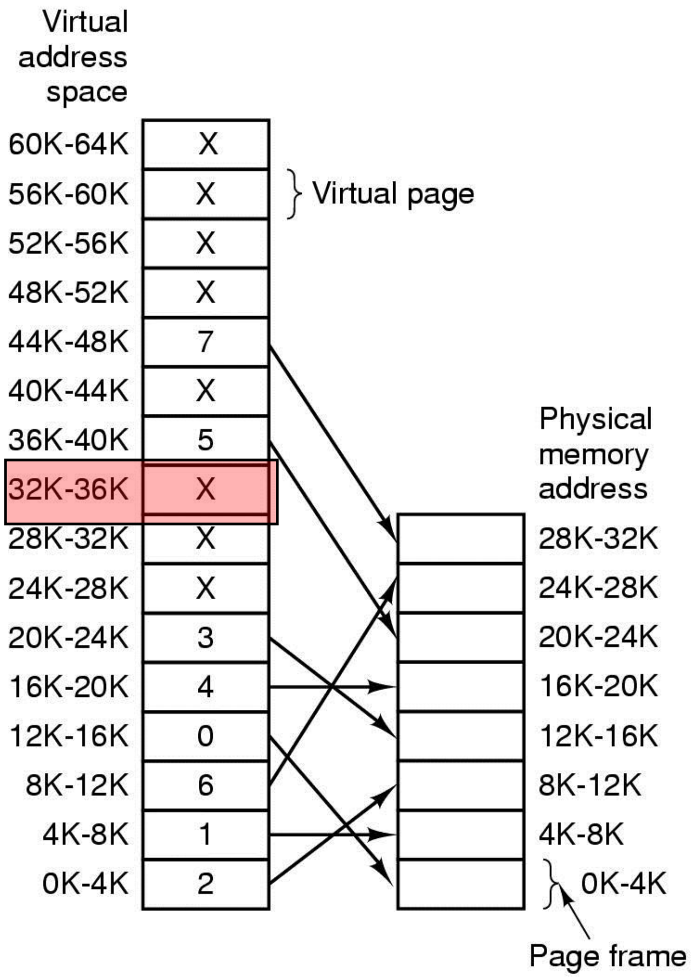
- **MOV REG, 20500**
  Virtual address 20500 (20480 + 20)
  Page frame 3
  Physical address 20+12288



Since virtual memory >> physical memory

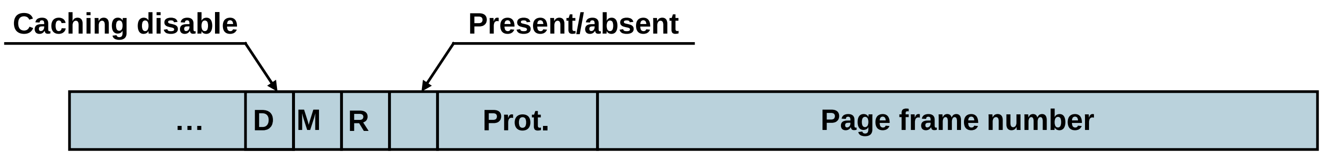
- Use a present/absent bit
- MMU checks –
  - If not there, “page fault” to the OS (trap)
  - OS picks a victim (?)
  - … sends victim to disk
  - … brings new one
  - … updates page table

MOVE REG, 32780
Virtual address 32780
Virtual page 8, byte 12 (32768+12)
Page is unmapped – page fault!



Page table entry

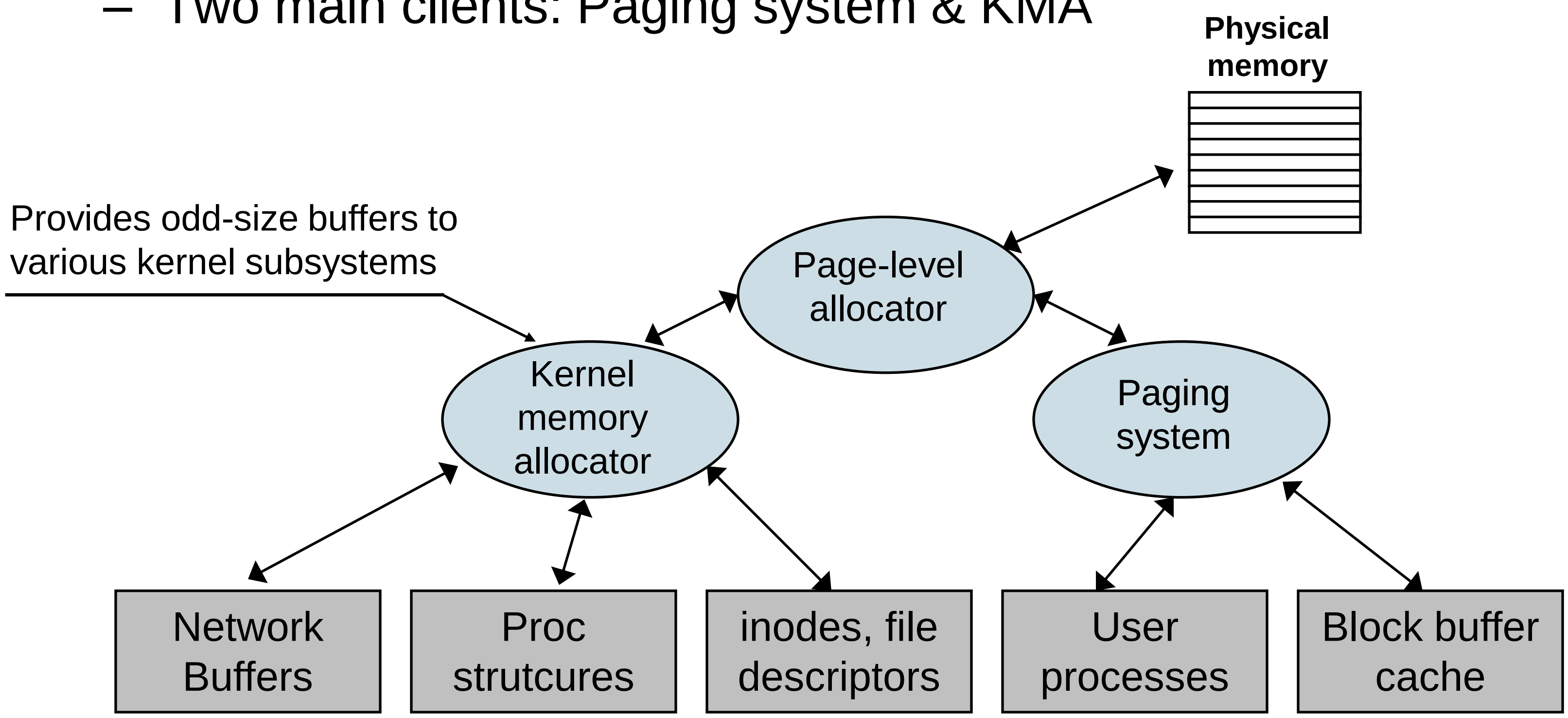
- An opportunity – there's a PTE lookup per memory reference, what else can we do with it?
- Looking at the details



  - Page frame number – the most important field
  - Protection – 1 bit for R&W or R or 3 bits for RWX
  - Present/absent bit
    - Says whether or not the virtual address is used
  - Modified (M): dirty bit
    - Set when a write to the page has occurred
  - Referenced (R): Has it being used?
  - To ensure we are not reading from cache (D)
    - Key for pages that map onto device registers rather than memory

Kernel memory allocation

- Most OS manage memory as set of fixed-size pages
- Kernel maintains a list of free pages
- Page-level allocator has
  - Two main routines: e.g `get_page()` & `freepage()` in SVR4
  - Two main clients: Paging system & KMA



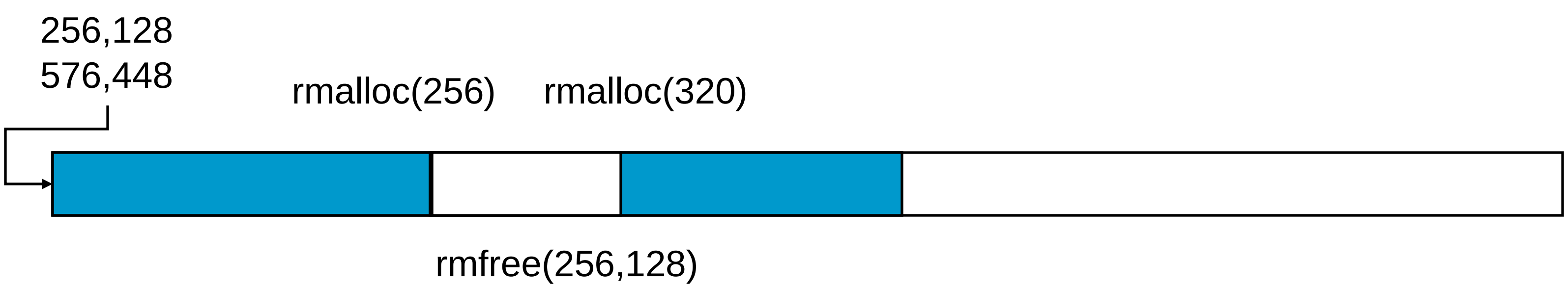
Kernel memory allocation

- KMA's common users
  - The pathname translation routine
  - Proc structures, vnodes, file descriptor blocks, …
- Since requests << page → page-level allocator is inappropriate
- KMA & the page-level allocator
  - Pre-allocates part of memory for the KMA
  - Allow KMA to request memory
  - Allow two-way exchange with the paging system
- Evaluation criteria
  - Memory utilization – physical memory is limited after all
  - Speed – it is used by various kernel subsystems
  - Simple API
  - Allow a two-way exchange with page-level allocator

KMA – Resource map allocator

- Resource map – a set of <base, size> pairs
- Initially the pool is described by a single pair
- … after a few exchanges … a list of entries per contiguous free regions
- Allocate requests based on
  - First fit, Best fit, Worst fit
- A simple interface

```
offset_t rmalloc(size);
void rmfree(base, size);
```

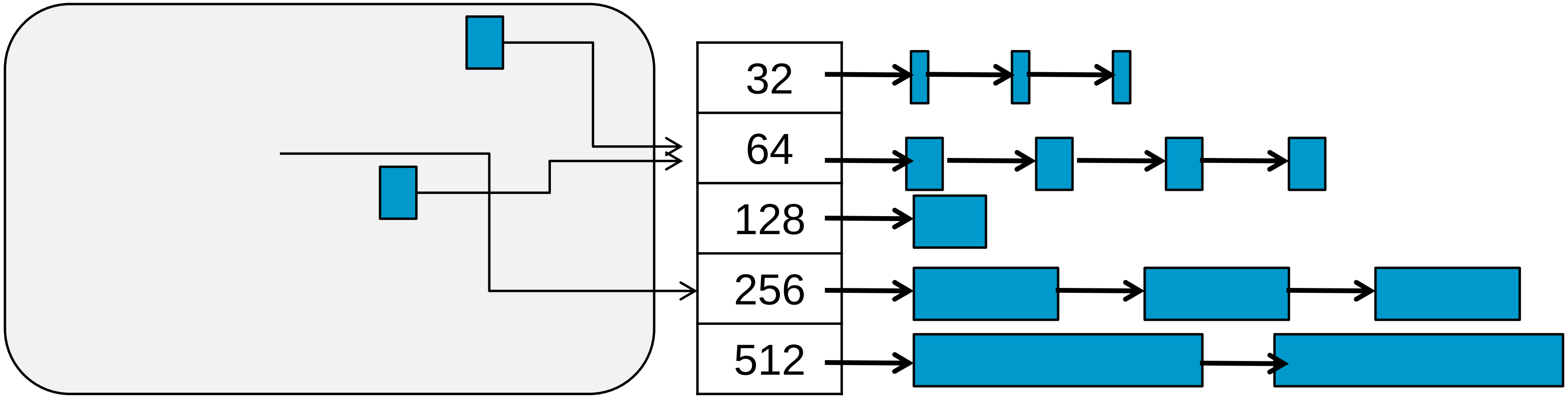


Resource map allocator

- Pros
  - Easy to implement
  - Not restricted to memory allocation
  - It avoid waste (although normally rounds up requests sizes for simplicity)
  - Client can release any part of the region
  - Allocator coalesces adjacent free regions
- Cons
  - After a while maps ended up fragmented – low utilization
  - Higher fragmentation, longer map
  - Map may need an allocator for its own entries
    - *How would you implement it?*
  - To coalesce regions, keep map sorted – expensive
  - Linear search to find a free region large enough

KMA – Simple power-of-two free list

- A set of free lists
- Each list keeps free buffers of a particular size (2^x)
- Each buffer has one word header
  - Pointer to next free buffer, if free or to
  - Pointer to free list (or size), if allocated



KMA – Simple power-of-two free list

- Allocating(size)
  - allocating (size + header) rounded up to next power of two
  - Return pointer to first byte *after* header
- Freeing doesn't require size as argument
  - Move pointer back header-size to access header
  - Put buffer in list
- Initialize allocator by preallocating buffers or get pages on demand; if it needs a buffer from an empty list …
  - Block request until a buffer is released
  - Satisfy request with a bigger buffer if available
  - Get a new page from page allocator

Power-of-two free lists

- Pros
  - Simple and pretty fast (avoids linear search)
  - Familiar programming interface (malloc, free)
  - Free does not require size; easier to program with
- Cons
  - Rounding means internal fragmentation
  - As many requests are power of two and we loose header; a lot of waste
  - No way to coalesce free buffers to get a bigger one
  - Rounding up may be a costly operation

Coming up ...

- The nitty-gritty details of virtual memory …