Memory Management

Today

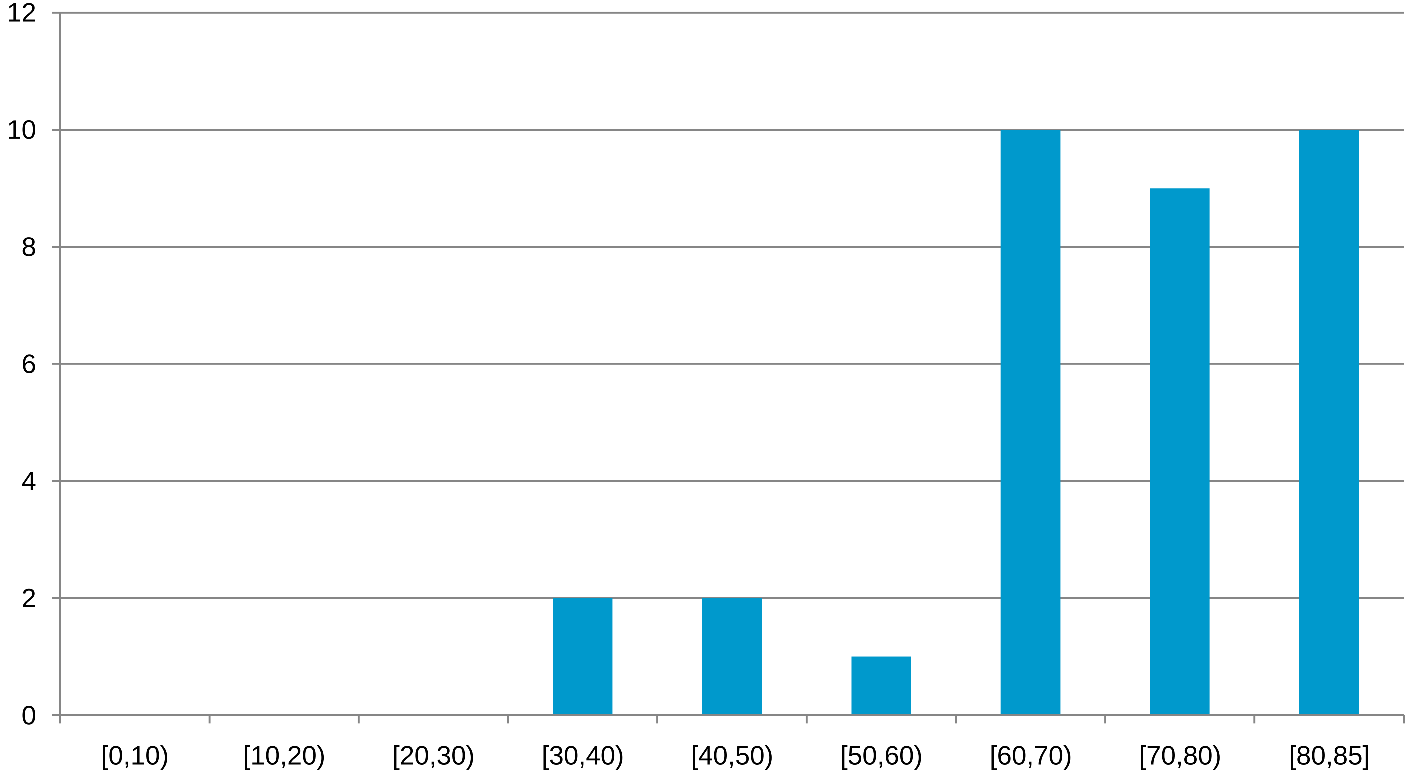
- Basic memory management
- Swapping
- Kernel memory allocation

Next Time

- Virtual memory

Midterm results

- Average 68.9705882
- Median 70.5
- Std dev 13.9576965



Schedule for the remaining weeks

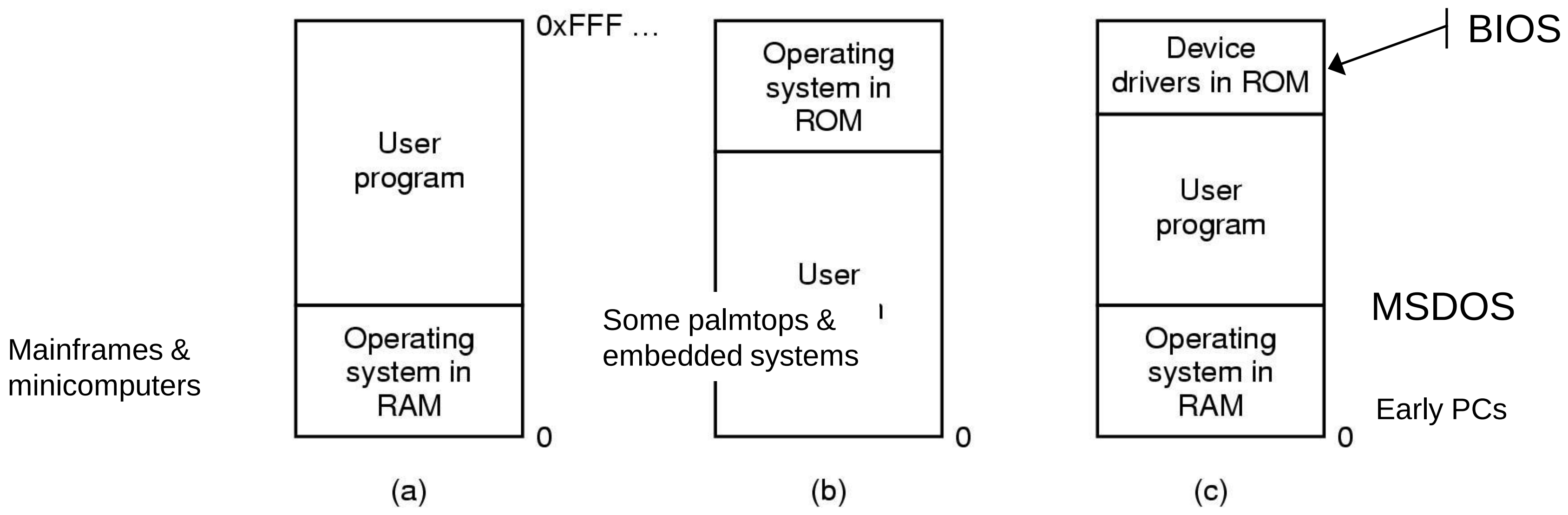
- 11/04 Memory mgmt.
  - *Project 3 out*
- 11/06 Memory mgmt
  - *Homework 3 out*
- 11/11 Virtual mem
- 11/13 Virtual mem
  - *Homework 3 in*
- *11/17 Project 3 In*
- 11/18 Mass-storage & I/O Systems
- 11/20 File systems I/F
  - *Homework 4 out*
  - *Project 4 out*
- 11/25 File systems impl.
- 11/27 Thanksgiving
- 12/02 Protection & security
  - *Project 4 in*
- *12/03 Homework 4 in*
- 12/05 Research in OS & Review
  - *Homework 5 in class*
- 12/09 Final

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

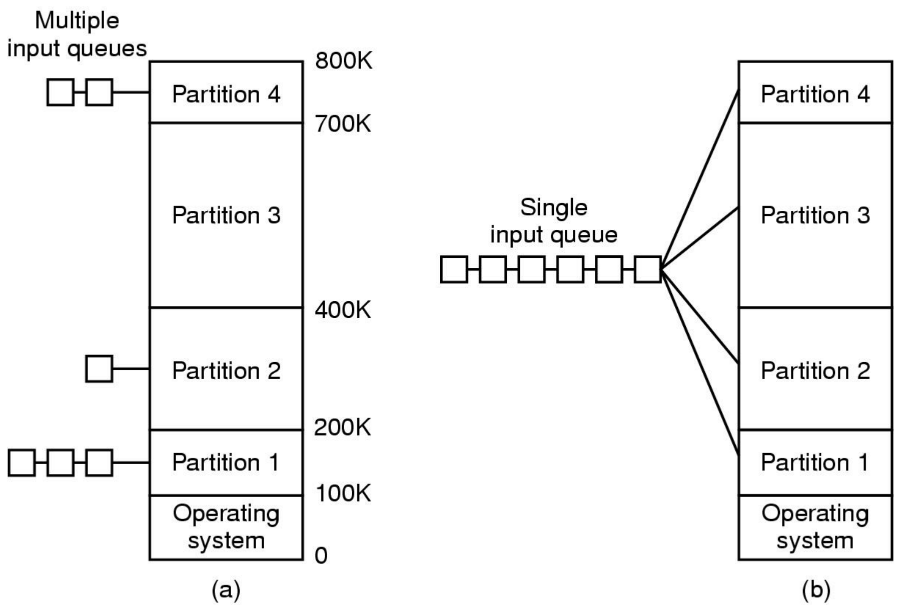
Basic memory management

- Simplest memory abstraction – no abstraction at all
  - Early mainframes (before '60), minicomputers (before '70) and PCs (before '80)
  - Only one program running at a time
  - Some alternatives for organizing memory



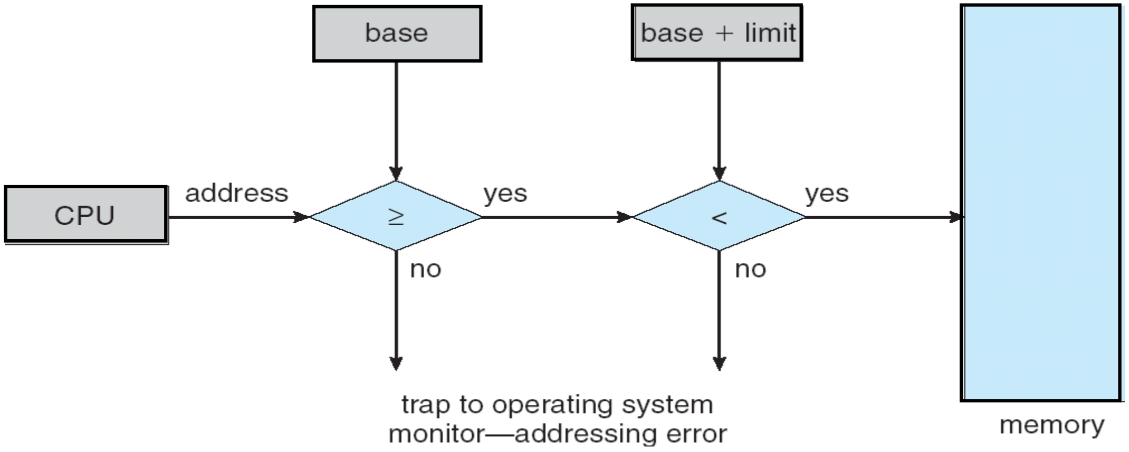
Multiprogramming w/ fixed partitions

- Multiprogramming – when one process is waiting for I/O, another one can use the CPU
- Two simple approaches
  - 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



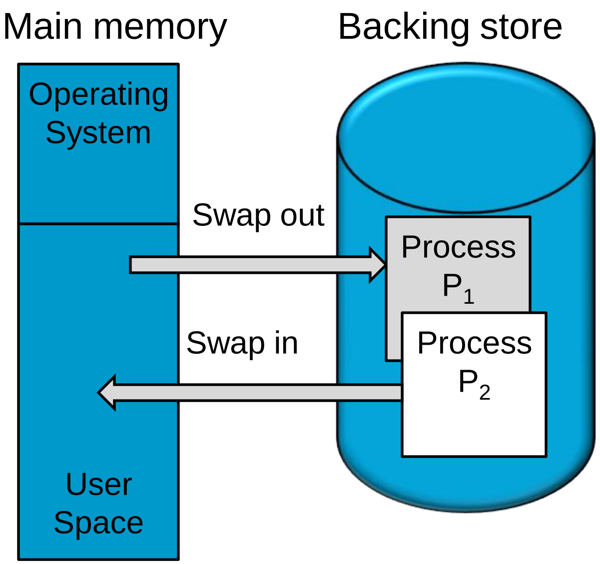
Two problems w/ multiprogramming

- Relocation and protection
  - Don't know where program will be loaded in memory
    - Address locations of variables & code routines
  - Keep a process out of other processes' partitions
- IBM OS/MFT - modify instructions on the fly; split memory into 2KB blocks & add key/code combination
- Use base and limit values (CDC 6600 & Intel 8088)
  - address locations + base value → physical address



Swapping

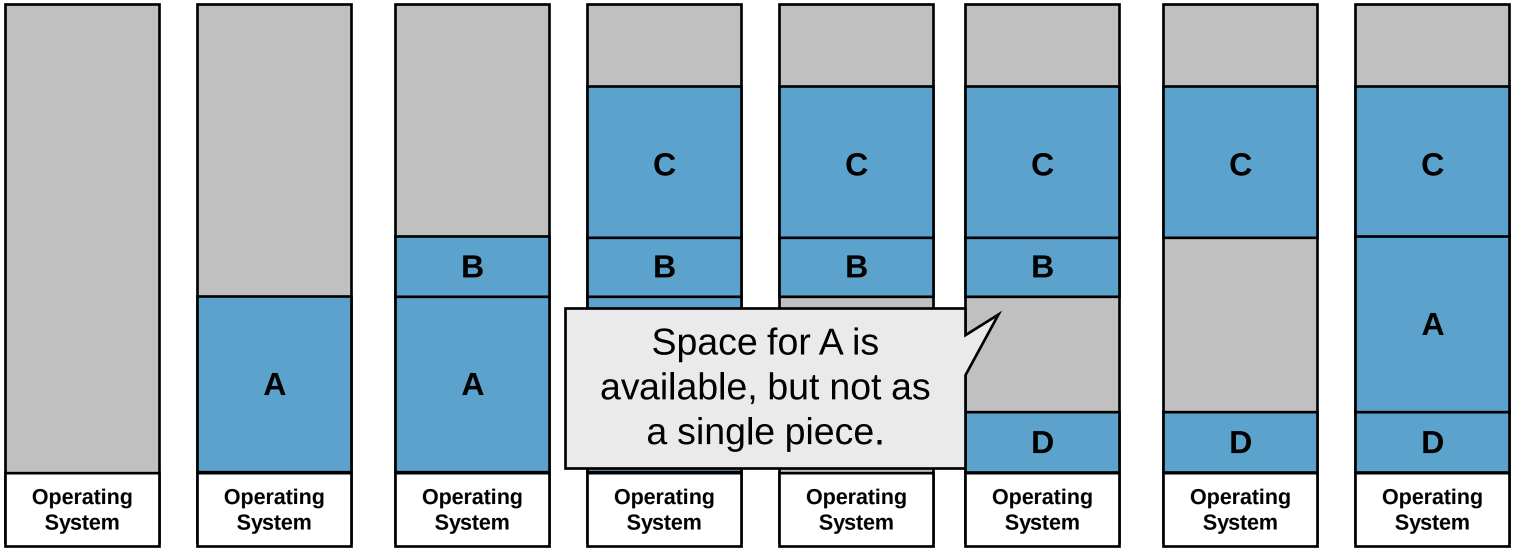
- Not enough memory for all processes?
  - Swapping
    - Simplest
    - Bring each process entirely
    - Move another one to disk
    - Compatible Time Sharing System (CTSS) – a uniprogrammed swapping system



  - 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 …

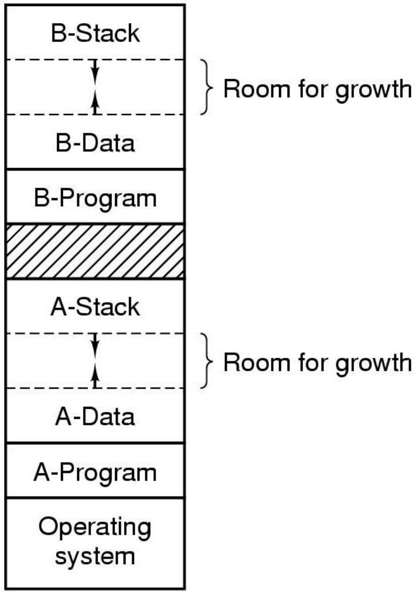


Fragmentation

- 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 (256MB machine, moving at 4B per 40 nanosec. ~ 2.7sec!)

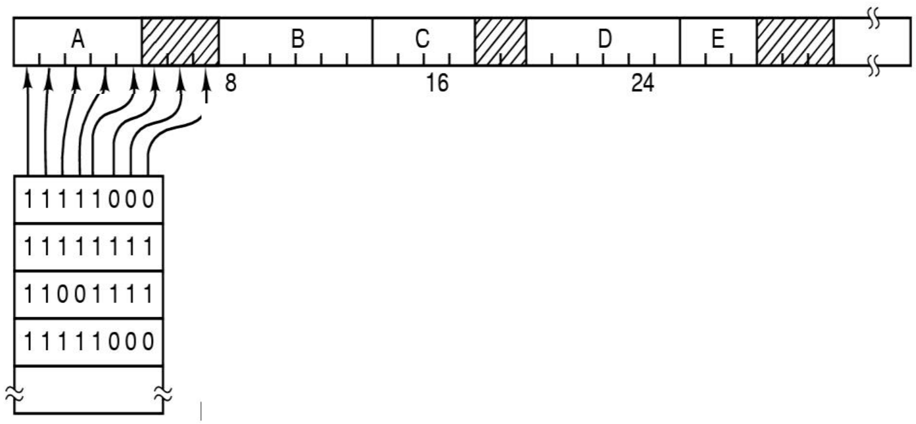
How much memory to allocate?

- 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



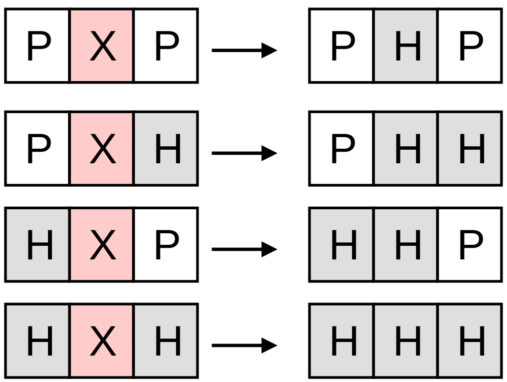
Memory management

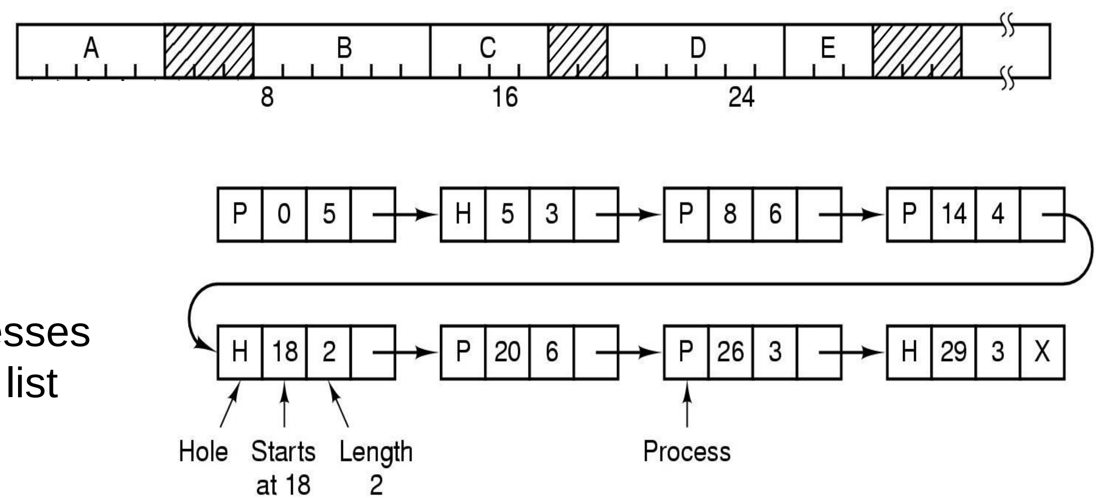
- With dynamically allocated memory
  - OS must keep track of allocated/free memory
  - Two general approaches - bit maps and linked lists
- Bit maps
  - Divide memory into allocation units
  - For each unit, a bit in the bitmap
  - Design issues - Size of allocation unit
    - The smaller the size, the larger the bitmap
    - The larger the size, the bigger the waste
  - Simple, but slow
    – find a big enough chunk?



Memory management with lists

- Linked list of allocated/free space
- List ordered by address
- Double link will make your life easier
  - Updating when a process is swapped out or terminates





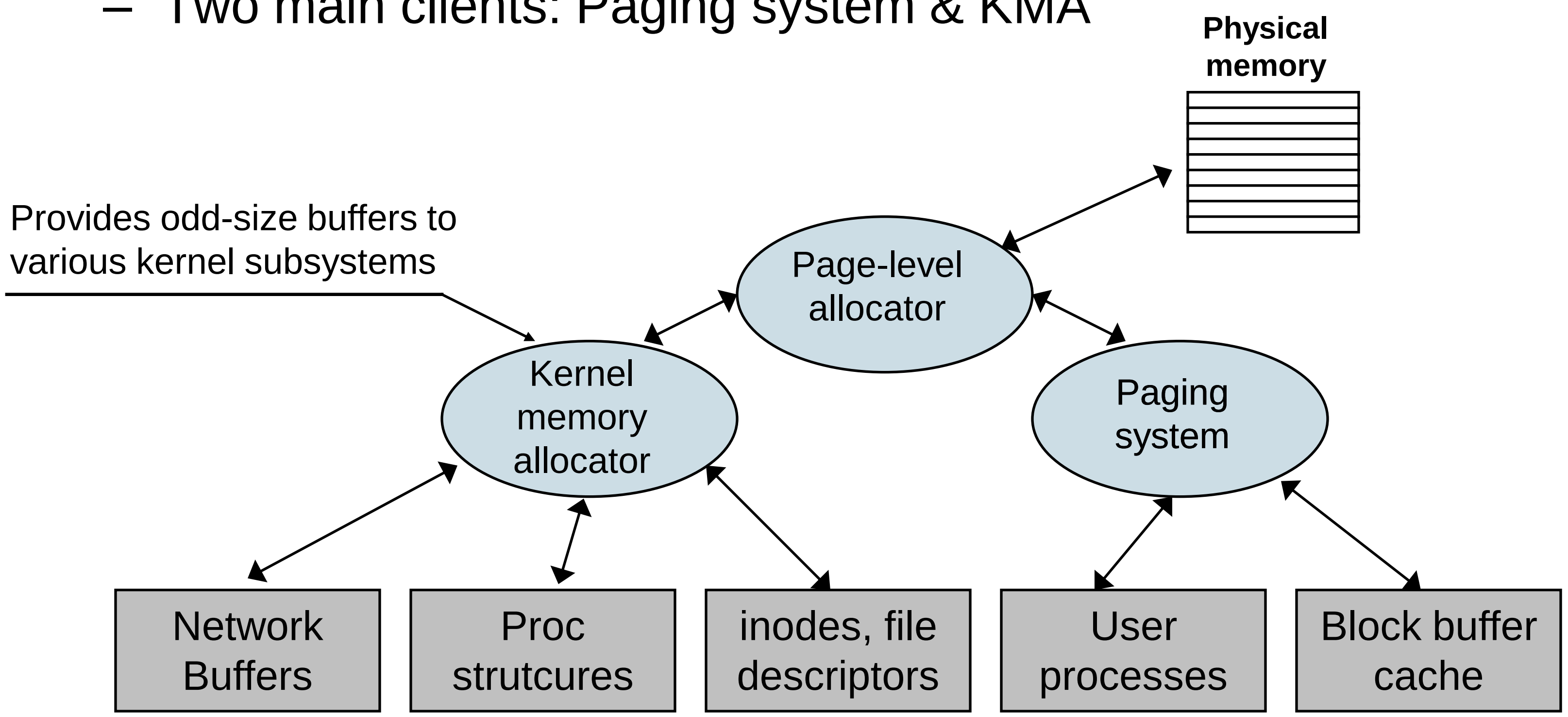
Keeping track of processes and holes in the same list

Picking a place – different algorithms

- First fit – simple and fast
- Next fit - ~ First fit but start where it left off
  - Worst performance than First fit
- Best fit – try to waste the least
  - More waste in tiny holes!
- Worst fit – try to "waste" the most
  - Not too good either
- Speeding things up
  - Two lists (free and allocated) – slows down deallocation
  - Order the hole list – first fit ~ best fit
  - Use the same holes to keep the list
  - Quick fit – list of commonly used hole sizes
    - N lists for N different common sizes (4KB, 8KB, …)
    - Allocation is quick, merging is expensive

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



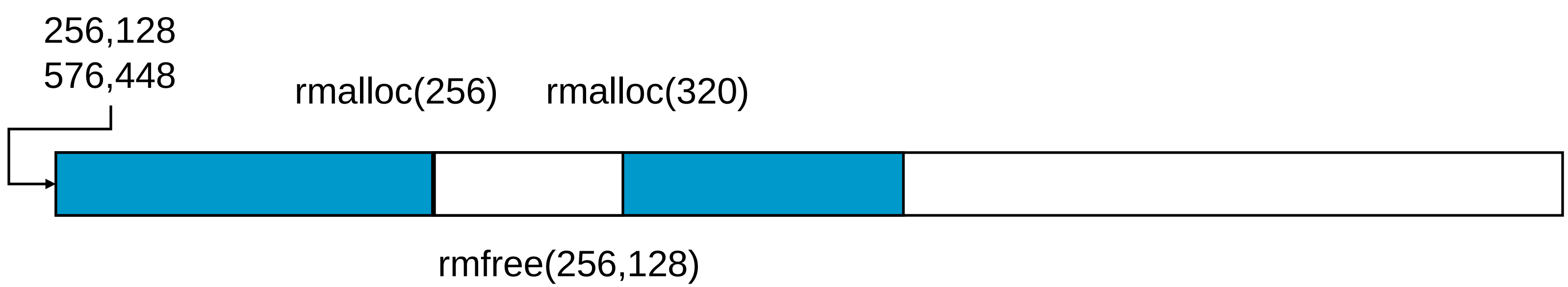
Kernel memory allocation

- 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
  - Utilization memory – 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);
```

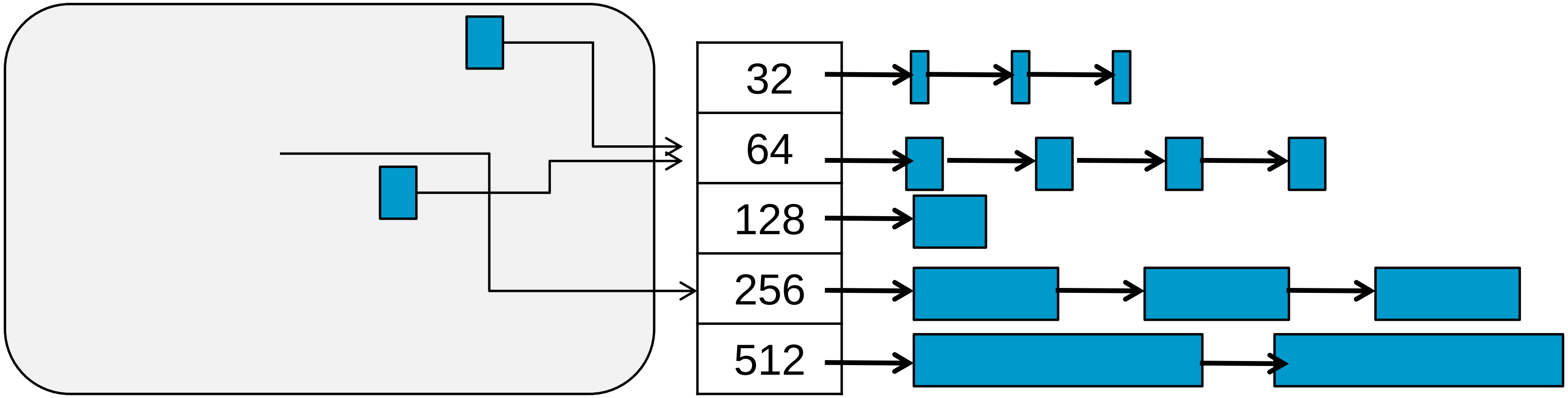


Resource map allocator

- 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 ...

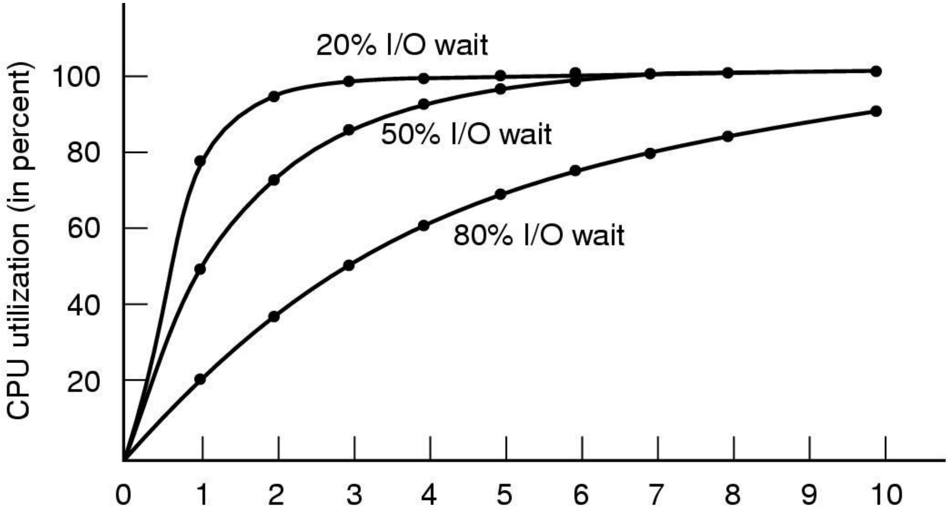
- The nitty-gritty details of virtual memory …

Modeling multiprogramming

- CPU utilization & multiprogramming
  - Utilization as a function of # of processes in memory
  - If process spends *p%* waiting for I/O

    Probability all processes waiting for I/O at once: p^n

    CPU Utilization $1 - p^n$



Performance of a MP system

Computer w/ 32MB

16MB for OS & 4 processes (@ 4MB per process)

- With 80% avg. waiting time
  - CPU Utilization – $1 - 0.8^4 = 1 - 0.41 = 0.6$: 60%
- Add 16MB – 4 more user processes
  - CPU Utilization – $1 - 0.8^8 = 0.83$: 83% … 38% increase
- Add 16MB – 4 more user processes
  - CPU Utilization – $1 - 0.8^{12} = 0.93$: 93% … 12% increase