Previous lecture review

- efficient memory management is needed in various areas
- user process space
  - internal - inside a process
    - in stack segment
    - in heap segment
  - external - between user processes
- kernel memory management

Advanced Memory Management Techniques

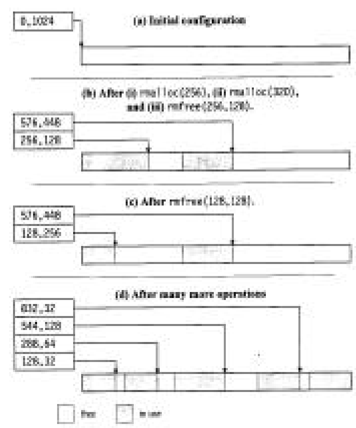
- Static vs. dynamic allocation
- resource map allocation
- power-of-two free list allocation
- buddy method allocation
- lazy buddy method allocation

What is there to manage?

- The kernel manages physical memory for both user processes and itself
  - user processes - virtual memory/paging (next lecture)
  - kernel needs such as:
    - process structures (PCBs/TCBs, etc.)
    - file system management structures management
    - network buffers and other communication structures for IPC
- The kernel subsystem that deals with kernel memory management is called *Kernel Memory Allocator* (KMA)
- first Unix kernels allocated the these structures statically; what's wrong with this approach?
  - can overflow
  - inflexible (cannot be adjusted to concrete system's needs)
  - conservative allocation leads to wasting memory
- need dynamic kernel memory allocation!

Resource map implementation of with FF, BF and WF

- The simplest dynamic memory allocation KMA uses *resource map*: a list of <base,size> where
  - base - start of free segment
  - size - size of free segment
- KMA can use either of
  - first fit
  - best fit
  - worst fit
- Unix uses FF for kernel buffers



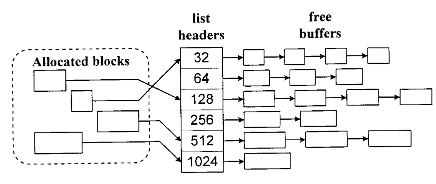
Analysis of resource map KMA

- advantages:
  - easy to implement
  - can allocate precise memory regions, clients can release parts of memory
  - adjacent free memory regions can be *coalesced* (joined) with extra work
- disadvantages:
  - the memory space gets fragmented
  - linear search for available memory space
  - resource map increases with fragmentation. what's wrong with that?
    - more kernel resource are used for the map
    - search time increases
  - to coalesce adjacent regions map needs to be sorted - expensive
  - hard to remove memory from the memory-mapped region

Power-of-two free list KMA

- A set of free buffer lists - each a power of two a.i. 32, 64, 128 … bytes
- each buffer has a one word (4 bytes) pointer
  - when the buffer is free - the pointer shows the next free buffer
  - when the buffer is used - it points to the size of the buffer
- the memory allocation requests are rounded up to the next power of 2
- when allocated - the buffer is removed from the list
- when freed - the buffer is returned to the appropriate free buffer list
- when list is empty KMA either allocates a larger buffer or delays request

used in Unix to implement user-level malloc() and free()

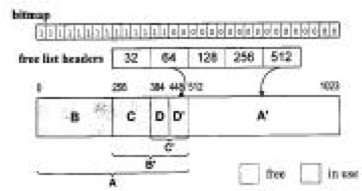


Analysis of power-of-two KMA

- Advantages:
  - simple and fast (bounded worst-case performance) - no linear searches
- Disadvantages:
  - cannot release parts of buffers
  - space is wasted on rounding to the next power of two
    - (what type of fragmentation is that?)
  - a word is wasted on the header - big problem for the memory requests that are power-of-two
  - can't coalesce adjacent free buffers

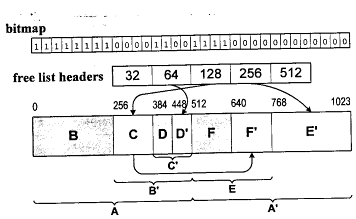
Buddy KMA

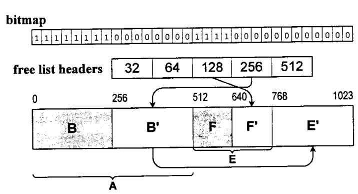
- Combines buffer coalescing with power-of-two allocator
- small buffers are created by (repeatedly) halving a large buffer
- when buffer is split the halves are called *buddies*
  - maintains the bitmap for the minimum possible buffer; 1 - allocated 2 - free
  - maintains a list of buffer sized (powers of two)
- example, initially we have a block of 1024bytes
  - allocate(256) - block is split into buddies of size 512 bytes - A and A'
  - A is split into B and B' - size 256
  - B allocated



Buddy KMA(cont)

- Allocate(128) - finds 128-free list empty; gets B' from 256-list and splits it into C and C' - size 128; allocates C
- allocate(64) - finds 64-list empty, gets C' from 128-list; splits it into D and D' - size 64, allocates D (see picture on previous page)
- allocate(128) - removes A'; splits it into E and E'; splits E into F and F', allocates F
- release(C, 128) - see picture on top
- release(D, 64) - coalesce D, D' to get C', coalesce C' and C to get B'





Analysis of buddy KMA

- advantages:
  - coalescence possible
  - possible dynamic modification of allocation region
- disadvantages:
  - performance - coalescing every time possibly to split up again; coalescing is recursive!
  - no partial release

Lazy buddy KMA

- *coalescence delay* - time it takes to check if the buddy is free and coalesce
- buddy KMA - each release operation - at least one coalescence delay
- if we allocate and deallocate same-size buffers - inefficient
- solution: coalesce only as necessary
  - operation is fast when we don't coalesce
  - operation is extremely slow if we coalesce
- middle approach:
  - we free the buffer making it available for reuse but not for coalescing (not marked in bitmap)
  - coalescing is done depending on the number of available buffers of certain class:
    - many (lazy state) - no coalescing necessary
    - borderline (reclaiming state) - coalescing is needed
    - few (accelerated state) - KMA must coalesce fast