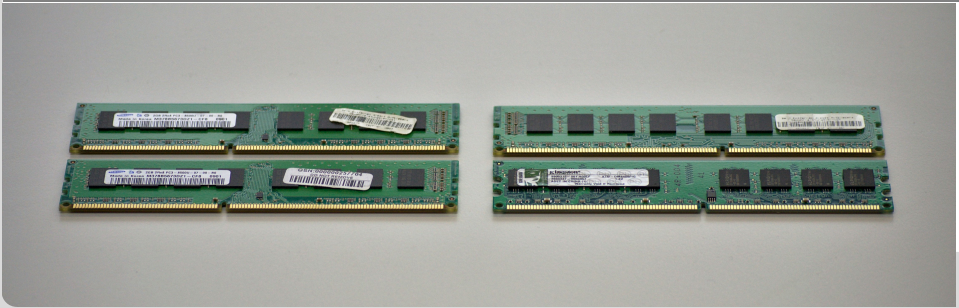
Karlsruhe Institute of Technology

Operating Systems

06. Dynamic Memory Allocation

Prof. Dr. Frank Bellosa | WT 2020/2021



Dynamic Memory Allocation

- Dynamic Memory Allocation: Allocate and free memory chunks of “arbitrary size” at arbitrary points in time
  - Almost every program uses it (heap)
  - Complex data structures don’t have to be specified statically
  - Data can grow as a function of input size
  - Kernel itself uses dynamic memory allocation for its data structures and buffers
- Implementation of dynamic memory allocation has a huge impact on performance
  - Both in user space and in kernel
- Common knowledge: It is impossible to construct a memory allocator that always performs well
  - “For any possible allocation algorithm, there exists a stream of allocation and deallocation requests that defeat the allocator and force it into severe fragmentation” [Rob71]

→ Picking a good allocation strategy requires a trade-off

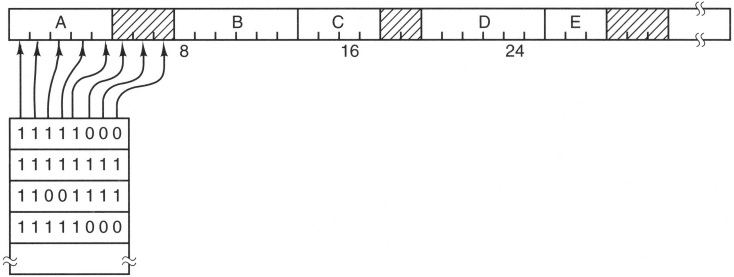
What does a Dynamic Memory Allocator do?

- Initially has a pool of free memory
- Needs to satisfy arbitrary `allocate` and `free` requests from that pool
- Cannot control the order or the number of requests
- Cannot move allocated regions (no compaction!)
  - Relocation is not possible (e.g., within a virtual AS)
  - Bad allocation decision is permanent!

  → Fragmentation is a core problem
- Needs to track which parts are in use and which parts are free

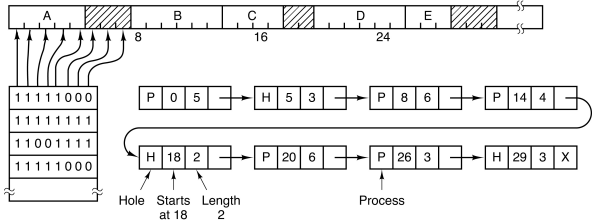
Bitmap

- Divide memory in allocation units of fixed size
- Use a bitmap to keep track if allocated (1) or free (0)
- Needs additional data structure to store allocation length
  - Otherwise cannot infer whether two adjacent allocations belong together or not from bitmap, but need this info for **`free`**. (e.g., try **`free`**ing address 8) [TB15]



List

- Either: Use one list-node for each allocated area
  - Needs extra space for the list
  - Allocation lengths already stored
- Or: Use one list-node for each unallocated area
  - Can keep list in the unallocated area (e.g., store size of free area and pointer to next free area in the free area itself)
  - Can search for free space with low overhead
- Or: Both (Double linked list speeds up merging after freeing) [TB15]



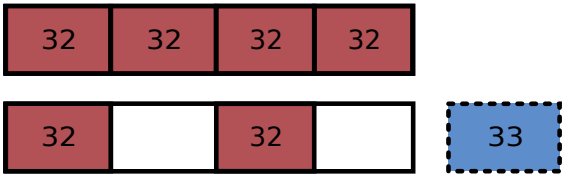
Why is Dynamic Memory Allocation Hard?

- Fragmentation is hard to handle
  - Fragmentation: Inability to use free memory
  - External fragmentation: Sum of free space is sufficient but cannot allocate sufficiently large contiguous block of free memory
  - Internal fragmentation: Overallocate resource requests to align memory blocks. Don't have free blocks left although there is sufficient unused memory within the blocks.

- Three factors required for fragmentation to occur
  - Different lifetimes (symmetric allocation times: no problem ⟶ stack)
  - Different sizes (same size: no problem ⟶ next allocation fits into any hole)
  - Inability to relocate previous allocations

A Pathological Allocation Example

- All memory is allocated in 32-byte chunks
- Every other allocation is released
- If it now wants to allocate 33-bytes:
  - The resource request fails...
  - ...although half of the memory is free



- Required "gross" memory in bad allocator: $\mathrm{M} \cdot \frac{\mathrm{n_{max}}}{\mathrm{n_{min}}}$
  - M = bytes of live data
  - $\mathrm{n_{min}}$ = smallest allocation, $\mathrm{n_{max}}$ = largest allocation
    use maximum size for any allocation

Best Fit vs. Worst Fit

- Idea: Keep large free memory chunks together for larger allocation requests that may arrive later
- Best-fit: Allocate the smallest free block that is large enough to fulfill the allocation request
  - Must search entire list, unless ordered by size
  - During free: coalesce adjacent blocks
- Problem: Sawdust
  - External fragments of allocations are so small that over time leftovers with unusable sawdust are everywhere
- Idea: Minimize sawdust by turning the strategy around
- Worst-fit: Allocate the largest free block
  - Must also search entire list, unless ordered by size
- In reality: Worse fragmentation than best-fit

First Fit

- Idea: If you produce fragmentation with best fit and worst fit alike, try to optimize for allocation speed
- First-fit: Allocate the first hole that is big enough
  - Fastest allocation policy
  - Produces leftovers of variable size
- Pathological case: Mix short lived 2n-byte allocations with long-lived (n+1)-byte allocations
  - Each time a large object is freed, a small chunk will be quickly taken, leaving a useless fragment
- In reality: Almost as good as best-fit

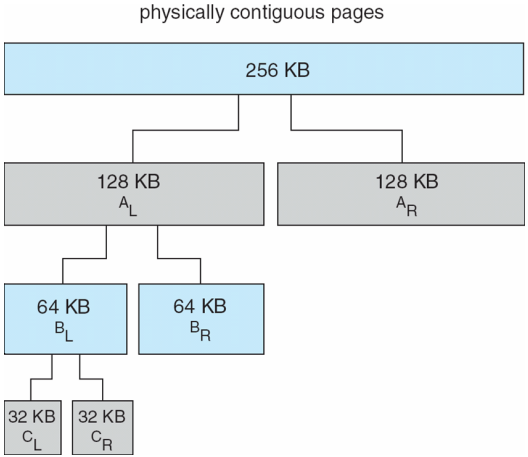
First Fit Nuances

- First-fit sorted by address order:
  - Blocks at front preferentially split, one at back only split when no larger one found before them
  - Seems to roughly sort free list by size
  - Similar to best fit
  - Sorting of list forces a large request to skip over many small blocks
- LIFO First-fit: Put object on front of list
  - Cheap & fast allocation policy
  - Hope same size used again (good cache locality)
- Next fit: Use First-fit, but remember where we found the last thing and start searching from there
  - Tends to break down entire list
  - Bad cache locality

Buddy Allocator

- Can be used to dynamically allocate contiguous chunks of fixed-size segments
  - e.g., used in the Linux kernel to allocate physical memory
- Allocates memory in powers of 2
  - All contiguous allocated/free memory chunks have fixed power-of-2 size
  - Request rounded up to next-higher power of 2
  - All chunks are naturally aligned
    (i.e., their starting address is a multiple of their size)
- If no sufficiently small memory block is available
  - Select larger available chunk and split it into two equal-sized “buddies”
  - Continue until appropriately sized chunk is available
- If two buddies are both free
  - Merge buddies to larger chunk encompassing both buddies

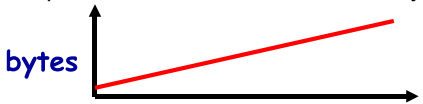
Buddy System (2)



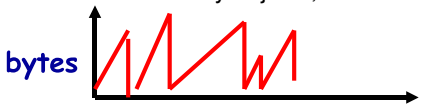
[SGG12]

Known patterns of real programs [Maz20]

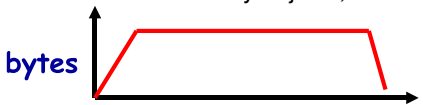
- Most real programs exhibit 1 or 2 (or all 3) of the following patterns of alloc/dealloc:
  - *Ramps*: accumulate data monotonically over time



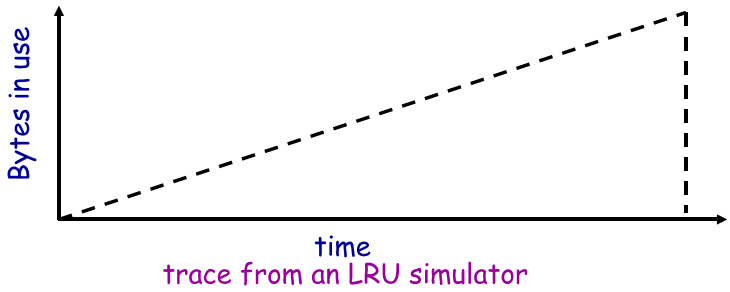
  - *Peaks*: allocate many objects, use briefly, then free all



  - *Plateaus*: allocate many objects, use for a long time

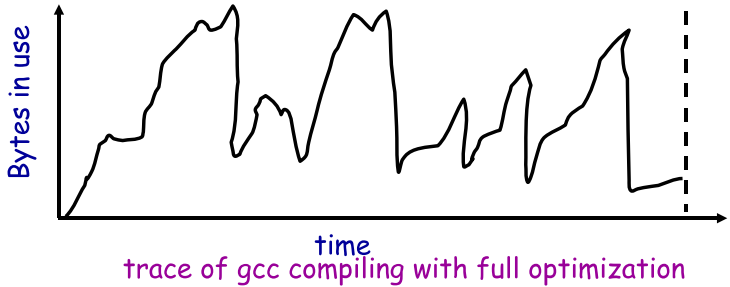


Pattern 1: Ramps [Maz20]



- In a practical sense: ramp = no free!
  - Implication for fragmentation?
  - What happens if you evaluate allocator with ramp programs only?

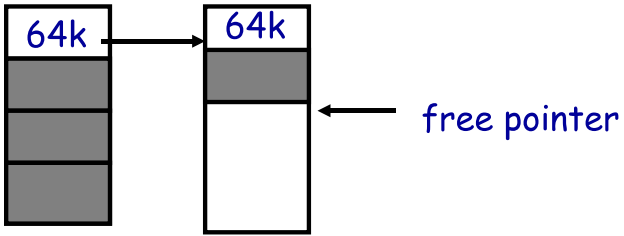
Pattern 2: Peaks [Maz20]



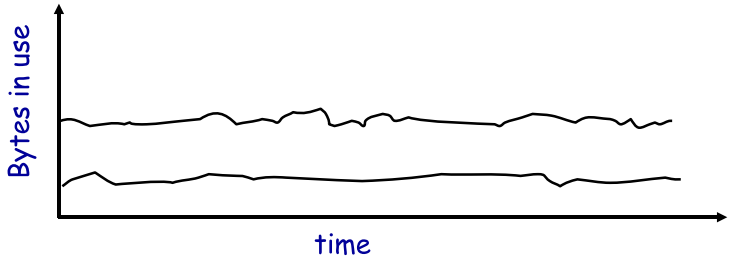
- Peaks: allocate many objects, use briefly, then free all
  - Fragmentation a real danger
  - What happens if peak allocated from contiguous memory?
  - Interleave peak & ramp? Interleave two different peaks?

Exploiting Peaks [Maz20]

- Peak phases: alloc a lot, then free everything
  - So have new allocation interface: alloc as before, but only support free of everything
  - Called “arena allocation”, “obstack” (object stack), or `alloca` procedure call
- Arena = a linked list of large chunks of memory
  - Advantages: alloc is a pointer increment, free is “free”
    No wasted space for tags or list pointers



Pattern 3: Plateau [Maz20]

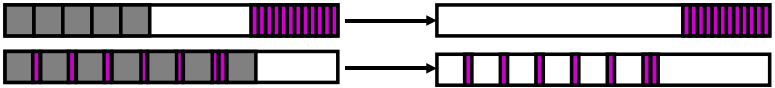


trace of perl running a string processing script

- Peaks: allocate many objects for a long time
  - What happens if overlap with peak or different plateau ?

Known patterns of real programs [Maz20]

- Segregation = reduced fragmentation:
  - Allocated at same time $\sim$ freed at same time
  - Different type $\sim$ freed at different time

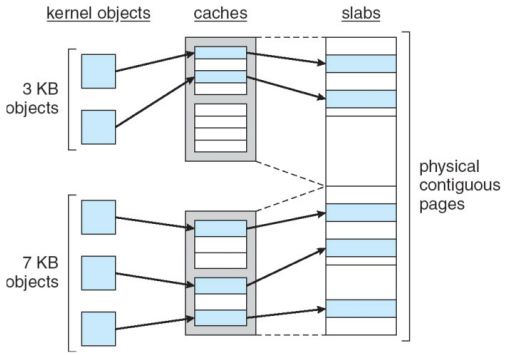


- Implementation observations :
  - Programs allocate small number of different sizes
  - Fragmentation at peak use more important than at low
  - Most allocations small ($<$ 10 words)

SLAB Allocator

- Kernel often allocates/frees memory for few, specific data objects of fixed size
- A **slab** is made up of multiple pages of contiguous physical memory
- A **cache** consists of one or multiple slabs
- Each cache stores only one kind of object (fixed size)
- Linux uses Buddy Allocator as underlying allocator for slabs

[SGG12]



References I

[Maz20] David Mazières. Cs140 – operating systems. *Stanford University*, 2020.

[Rob71] J. M. Robson. An estimate of the store size necessary for dynamic storage allocation. *J. ACM*, 18(3):416–423, July 1971.

[SGG12] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

[TB15] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 4th edition, 2015.