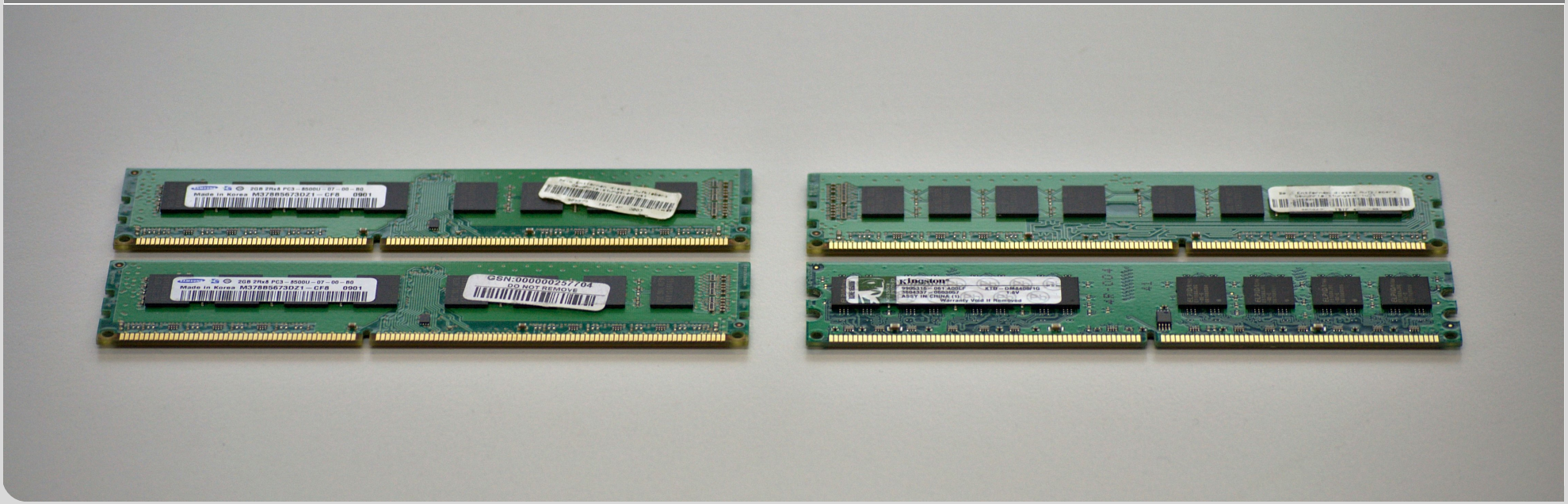
Karlsruhe Institute of Technology

Betriebssysteme

14. Memory Allocation

Prof. Dr.-Ing. Frank Bellosa | WT 2016/2017

KARLSRUHE INSTITUTE OF TECHNOLOGY (KIT) – OPERATING SYSTEMS GROUP



Dynamic Memory Allocation

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)
  - Don't have to statically specify complex data structures
  - Can have data grow as a function of input size
  - Kernel itself uses dynamic memory allocation for its data structures!
- Implementation of dynamic memory has huge impact on performance
  - Both in user space and in kernel
- Proven fact: 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" (Robson)
  - Need to understand the trade-offs to pick a good allocation strategy

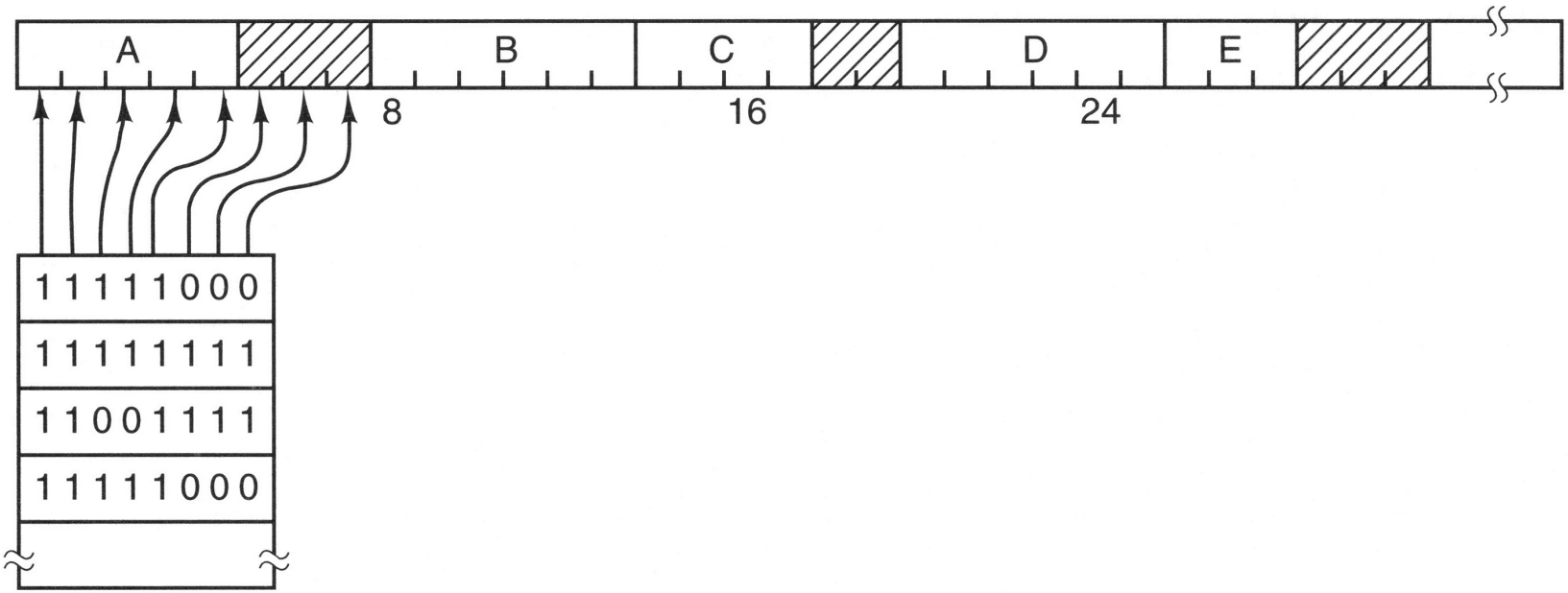
What does a Dynamic Memory Allocator do?

- Initially has a pool of free memory
- Needs to satisfy arbitrary `allocate` and `free` requests from that pool
- Needs to track which parts are in use and which parts are free
- 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 placement decision is permanent!

  → Fragmentation is a core problem

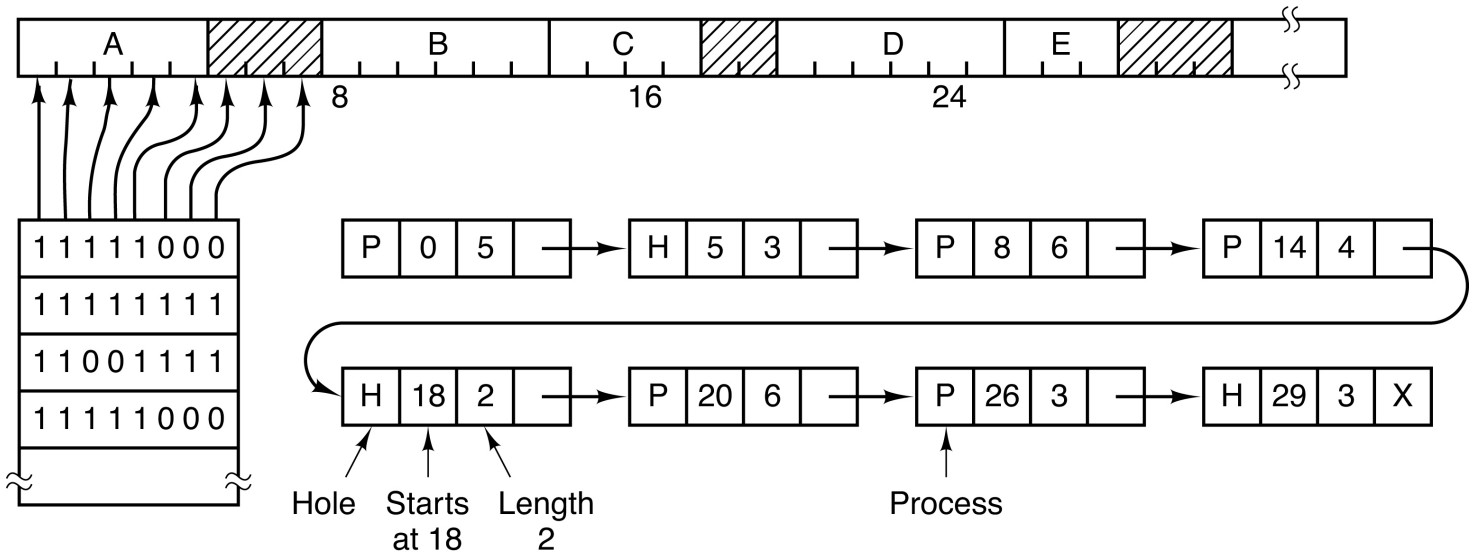
Bitmap

- 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)



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)
  - Needs additional data structure to store allocation lengths
  - Can search for free space with low overhead
- Or: Both



Why is Dynamic Memory Allocation Hard?

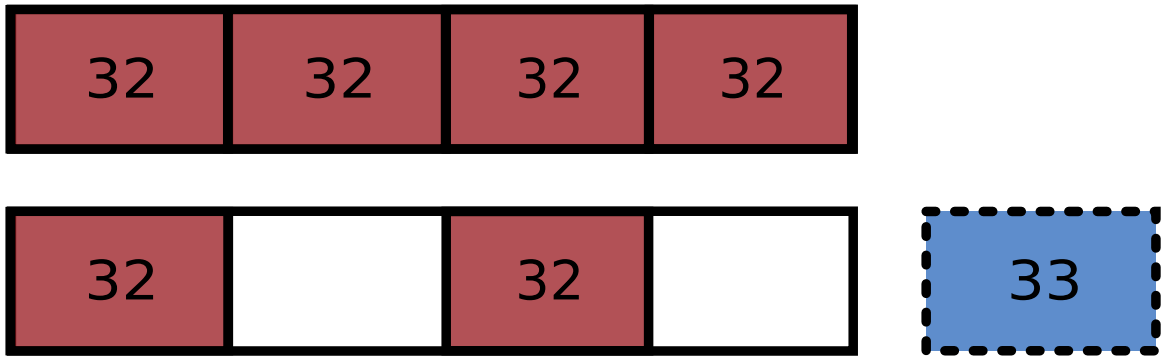
- Fragmentation is hard to handle (recall lecture 9)
  - Fragmentation: Inability to use free memory
  - External fragmentation: Sum of free space is sufficient but cannot allocate sufficiently large contiguous block of free memory
  - Internal fragmentation: Overallocate resource requests to align memory 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

- All three are present in dynamic memory allocators

A Pathological Allocation Example

- Say an application allocated all memory in 32-byte chunks
- It then releases every other allocation
- 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 size

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 store the allocation request
  - Must search entire list, unless ordered by size
  - During free: coalesce adjacent blocks
- Problem: Sawdust
  - Remainder so small that over time left with unusable sawdust 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 leftover holes 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

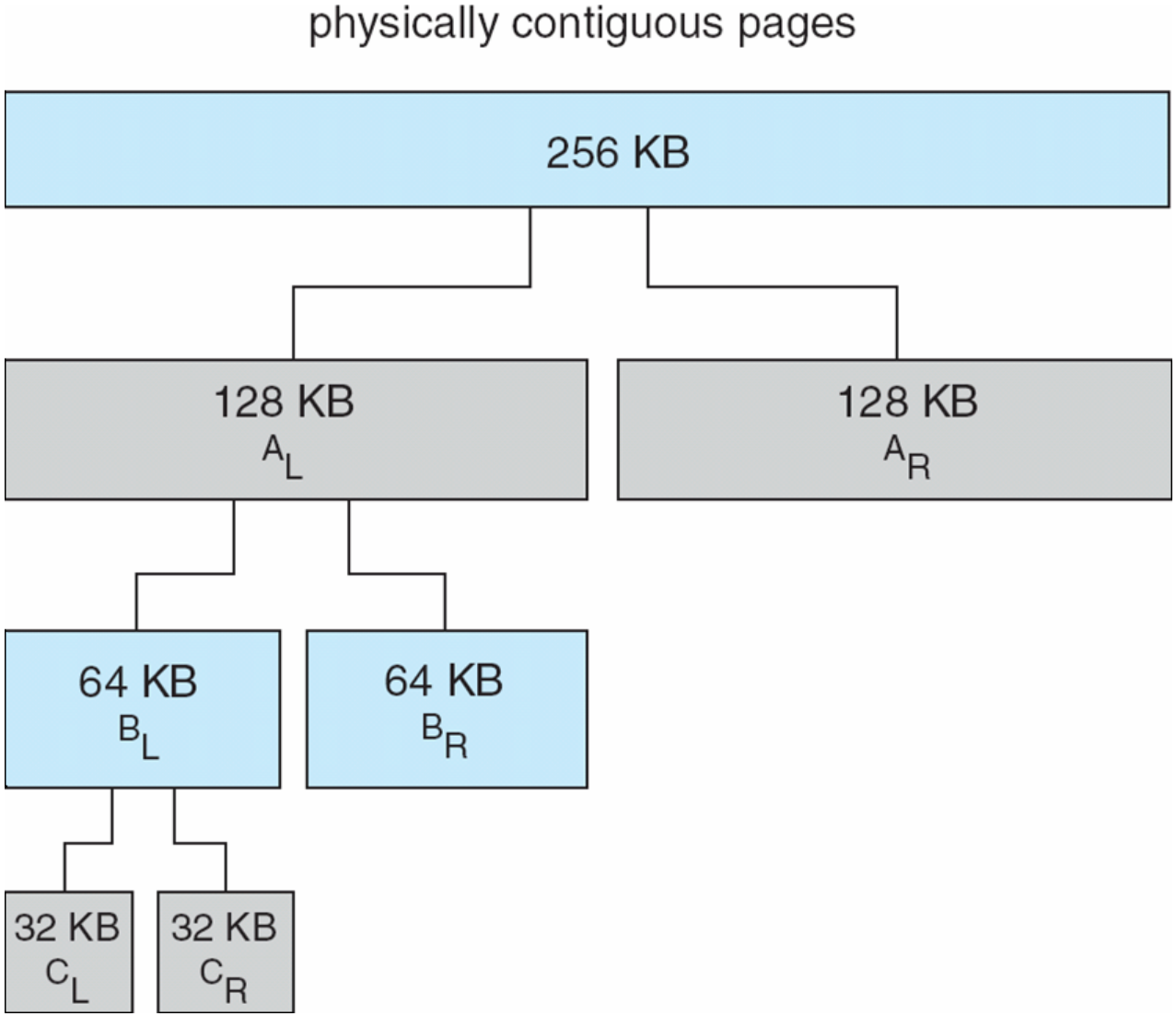
First Fit Nuances

- 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

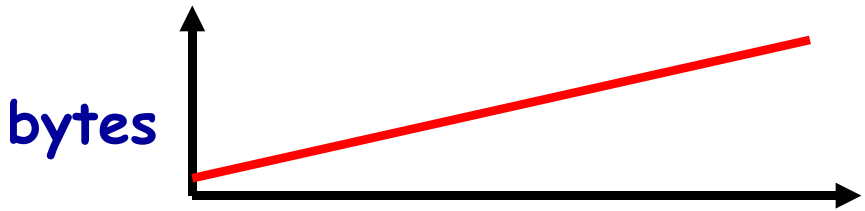
- Can be used to dynamically allocate contiguous chunks of fixed-size segments
  - e.g., Used in Linux kernel to allocate physical memory
- Allocates memory in powers of 2
  - All contiguous allocated/free memory chunks have fixed power-of-2 size
  - Request rounded up to next-higher power of 2
  - All 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)

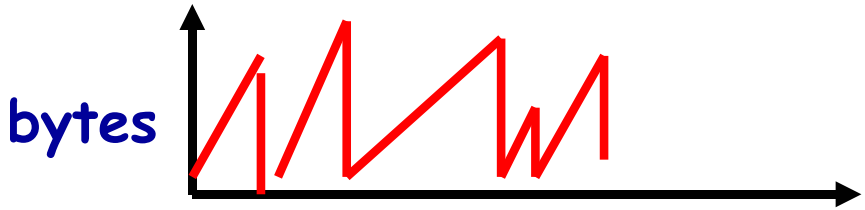


Known patterns of real programs

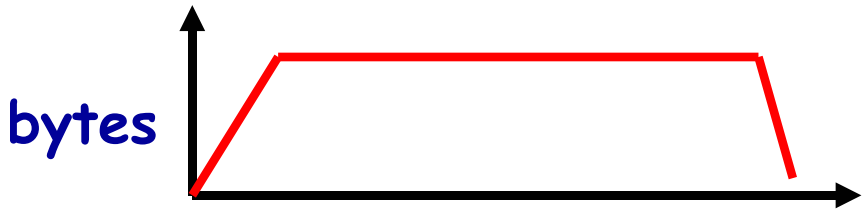
- So far we've treated programs as black boxes.
- Most real programs exhibit 1 or 2 (or all 3) of the following patterns of alloc/dealloc:
  - *Ramps*: accumulate data monotonically over time



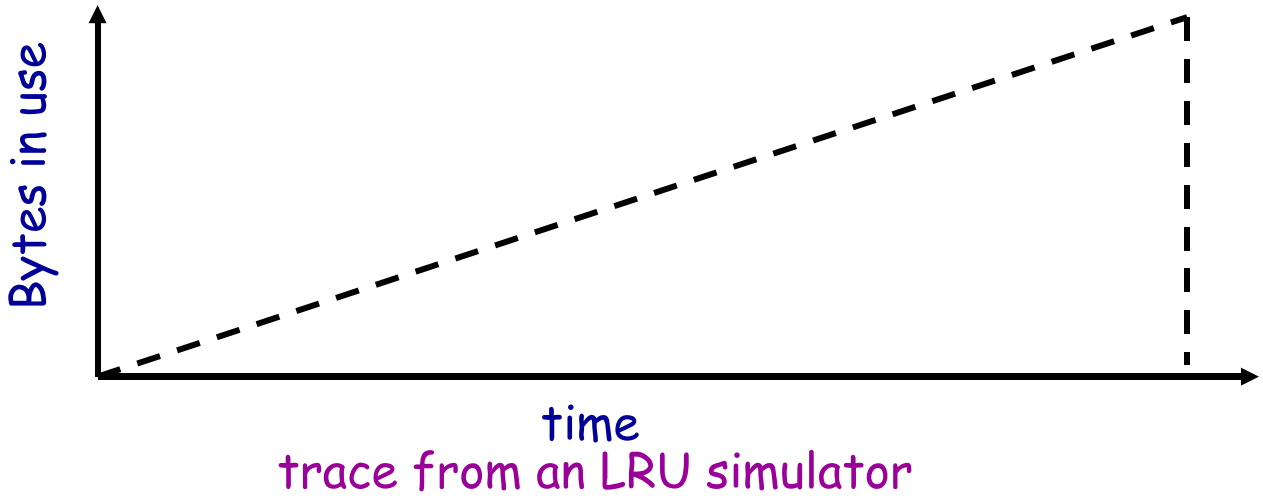
  - *Peaks*: allocate many objects, use briefly, then free all



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

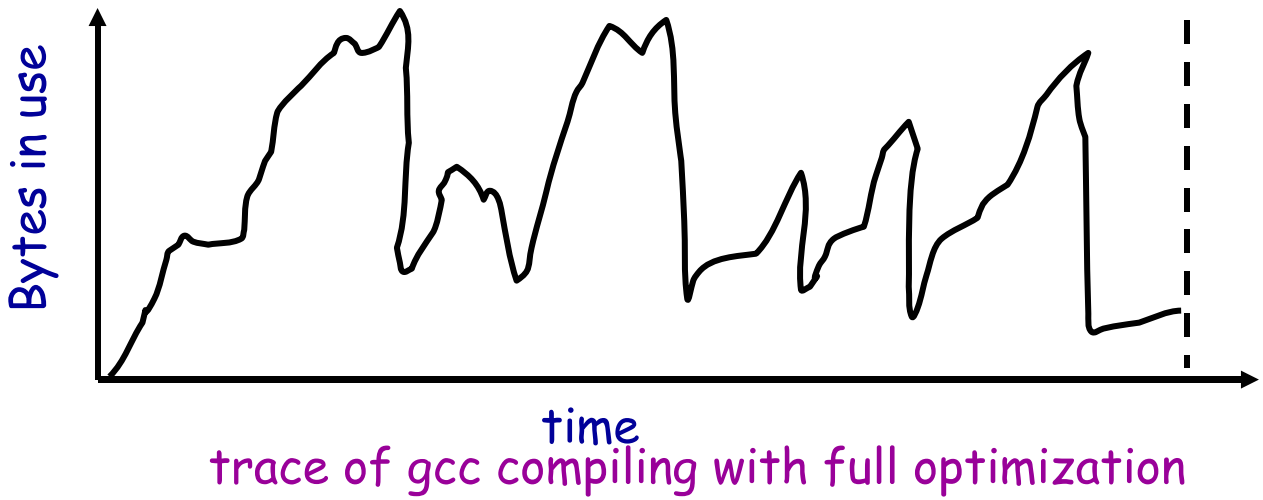


Pattern 1: Ramps



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

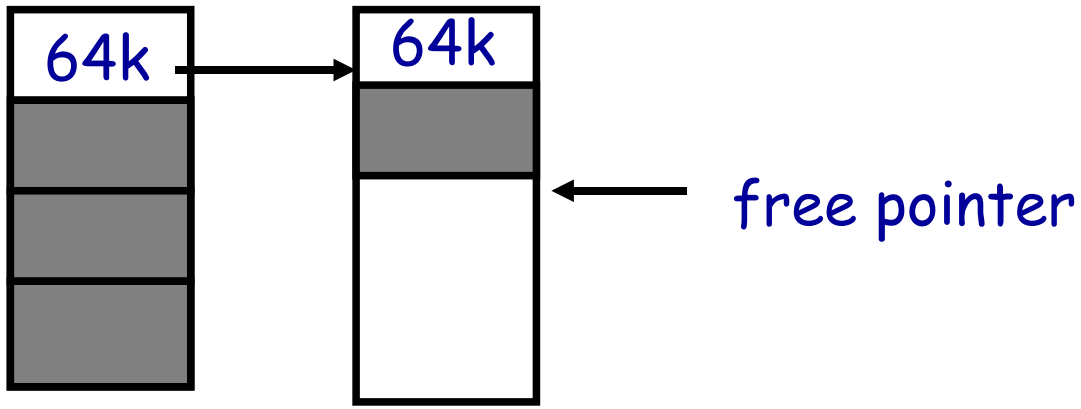
Pattern 2: Peaks



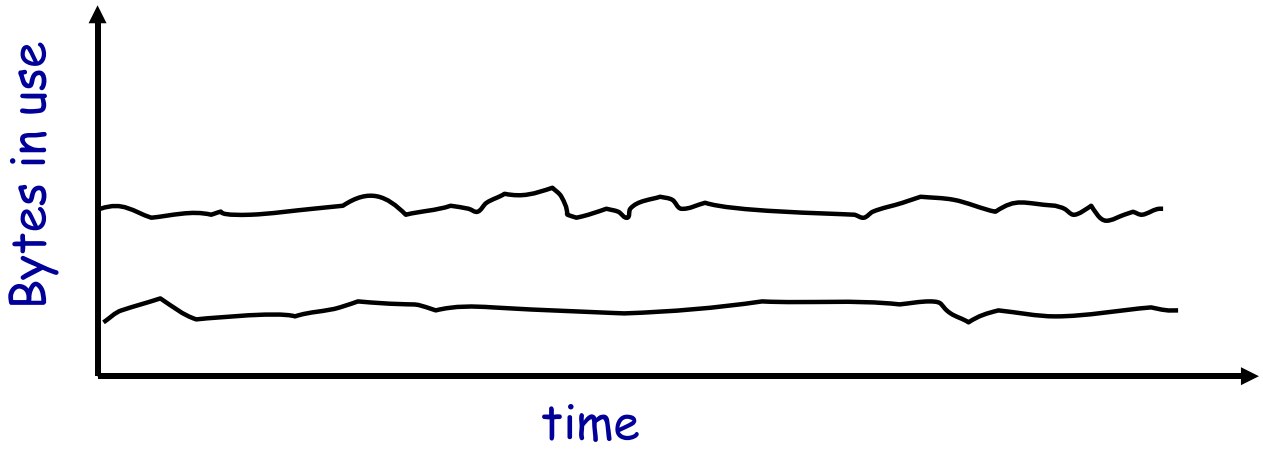
- Peaks: allocate many objects, use briefly, then free all
  - Fragmentation a real danger
  - What happens if peak allocated from contiguous memory?
  - Interleave peak & ramp? Interleave two different peaks?

Exploiting Peaks

- 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 (by compiler people)
- 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

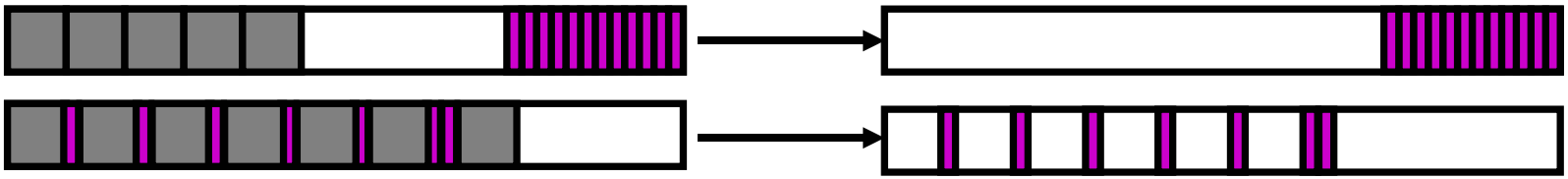


trace of perl running a string processing script

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

Known patterns of real programs

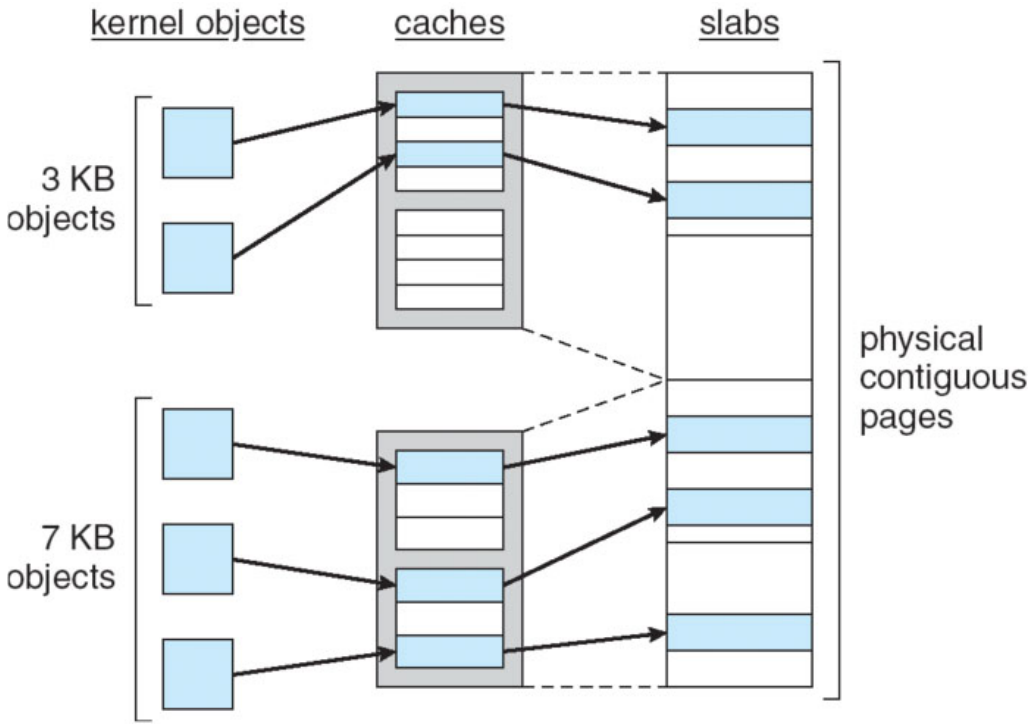
- Segregation = reduced fragmentation:
  - Allocated at same time $\sim$ freed at same time
  - Different type $\sim$ freed at different time



- 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



Summary

- Dynamic memory means allocating and freeing memory chunks of different sizes at any time
- It is impossible to construct a memory allocator that always performs well
- The main problem of dynamic memory allocators is fragmentation
- Typical dynamic memory data structures are bitmaps and free-lists
- Simple allocation strategies that perform reasonably well are: best-fit and first-fit
- More advanced strategies are the buddy- and slab allocator that are used in the Linux kernel to allocate page frames and in-kernel data structures

Further Reading

- Tanenbaum/Bos, “Modern Operating Systems”, 4th Edition:
  - Pages 190–194
  - Pages 761–763