CS 318 Principles of Operating Systems

Fall 2017

Lecture 13: Dynamic Memory Allocation

Ryan Huang

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

Administrivia

- **Lab 2 due Friday midnight**
- **Guoye will be traveling 10/21 to 10/30**
  - Lab 3 overview session will be livestreamed or recorded
  - His office hours will be canceled, but please ask questions on Piazza, via emails, or request remote meeting
- **Midterm grading**

Memory Allocation

- **Static Allocation (fixed in size)**
  - want to create data structures that are fixed and don't need to grow or shrink
  - global variables, e.g., char name[16];
  - done at compile time

- **Dynamic Allocation (change in size)**
  - want to increase or decrease the size of a data structure according to different demands
  - done at run time

Dynamic Memory Allocation

- **Almost every useful program uses it**
  - Gives wonderful functionality benefits
  - Don’t have to statically specify complex data structures
  - Can have data grow as a function of input size
  - Allows recursive procedures (stack growth)
  - But, can have a huge impact on performance

- **Two types of dynamic memory allocation**
  - Stack allocation: restricted, but simple and efficient
  - Heap allocation (focus today): general, but difficult to implement.

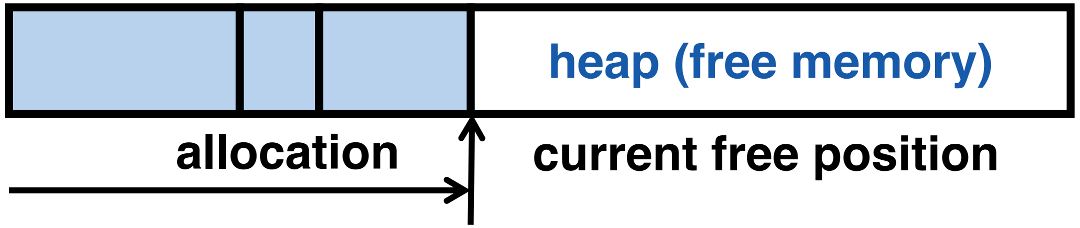
Dynamic Memory Allocation

- **Today: how to implement dynamic heap allocation**
  - Lecture based on [Wilson] (good survey from 1995)

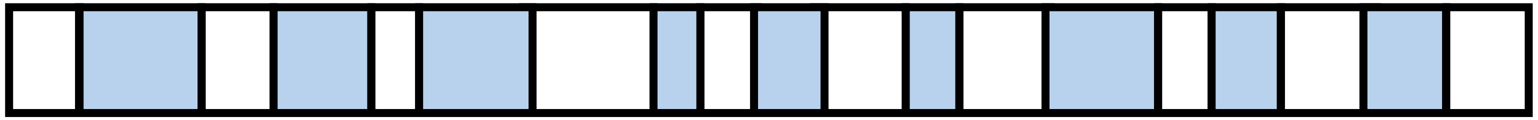
- **Some interesting facts:**
  - Two or three line code change can have huge, non-obvious impact on how well allocator works (examples to come)
  - Proven: impossible to construct an "always good" allocator
  - Surprising result: after 35 years, memory management still poorly understood
    - *Mallacc: Accelerating Memory Allocation:* ASPLOS 2017 Highlights
  - Big companies may write their own “malloc”
    - Google: TCMalloc
    - Facebook: jemalloc

Why Is It Hard?

- **Satisfy arbitrary set of allocation and frees.**
- **Easy without free: set a pointer to the beginning of some big chunk of memory ("heap") and increment on each allocation:**



- **Problem: free creates holes ("fragmentation") Result? Lots of free space but cannot satisfy request!**

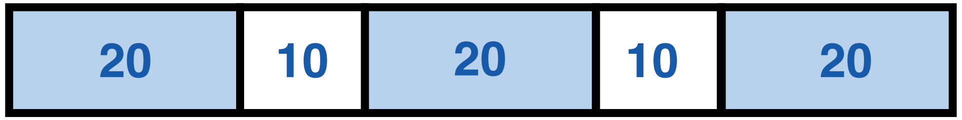


More Abstractly

freelist

- **What an allocator must do?**
  - Track which parts of memory in use, which parts are free
  - Ideal: no wasted space, no time overhead
- **What the allocator cannot do?**
  - Control order of the number and size of requested blocks
  - Know the number, size, & lifetime of future allocations
  - Move allocated regions (bad placement decisions permanent)

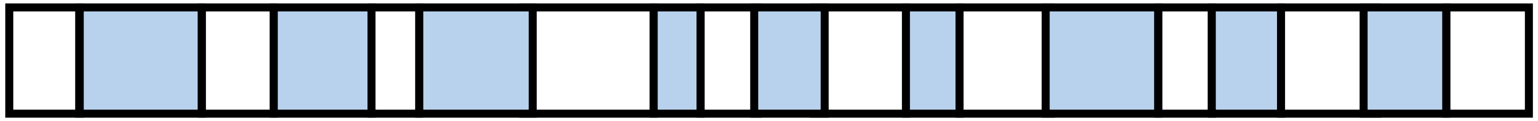
`malloc(20)?`



- **The core fight: minimize fragmentation**
  - App frees blocks in any order, creating holes in “heap”
  - Holes too small? cannot satisfy future requests

What Is Fragmentation Really?

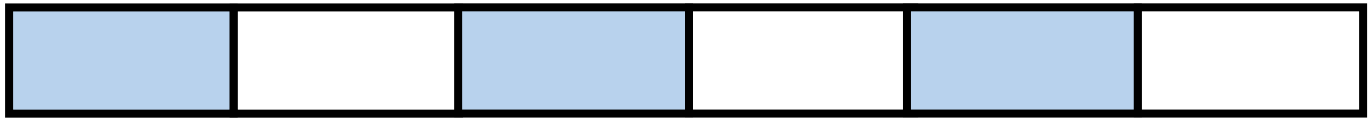
- **Inability to use memory that is free**
- **Two factors required for fragmentation**
  1. Different lifetimes—if adjacent objects die at different times, then fragmentation:

     - If all objects die at the same time, then no fragmentation:

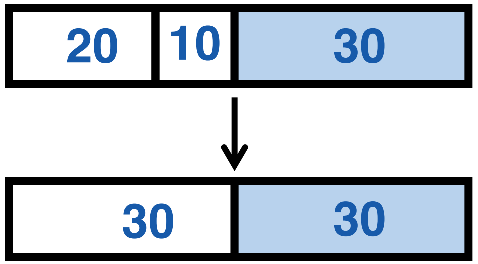
     

  2. Different sizes: If all requests the same size, then no fragmentation (that's why no external fragmentation with paging):

Important Decisions

- **Placement choice: where in free memory to put a requested block?**
  - Freedom: can select any memory in the heap
  - Ideal: put block where it won't cause fragmentation later (impossible in general: requires future knowledge)
- **Split free blocks to satisfy smaller requests?**
  - Fights internal fragmentation
  - Freedom: can choose any larger block to split
  - One way: choose block with smallest remainder (best fit)
- **Coalescing free blocks to yield larger blocks**
  - Freedom: when to coalesce (deferring can save work)
  - Fights external fragmentation



Impossible to “Solve” Fragmentation

- **If you read allocation papers to find the best allocator**
  - All discussions revolve around tradeoffs
  - The reason? There cannot be a best allocator
- **Theoretical result:**
  - For any possible allocation algorithm, there exist streams of allocation and deallocation requests that defeat the allocator and force it into severe fragmentation.
- **How much fragmentation should we tolerate?**
  - Let M = bytes of live data, $\mathrm{n_{min}}$ = smallest allocation, $\mathrm{n_{max}}$ = largest allocation
  - Bad allocator: $\mathrm{M} \cdot (\mathrm{n_{max}}/\mathrm{n_{min}})$
    - E.g., make all allocations of size $\mathrm{n_{max}}$ regardless of requested size
  - Good allocator: $\sim \mathrm{M} \cdot \log(\mathrm{n_{max}}/\mathrm{n_{min}})$

Pathological Examples

- **Suppose heap currently has 7 20-byte chunks**

20	20	20	20	20	20	20

  - What’s a bad stream of frees and then allocates?

- **Next: two allocators (best fit, first fit) that, in practice, work pretty well**
  - “pretty well” = ~20% fragmentation under many workloads

Pathological Examples

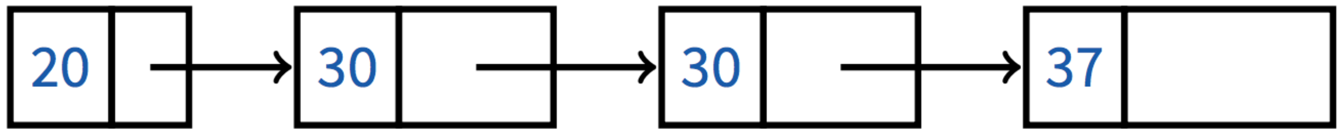
- **Suppose heap currently has 7 20-byte chunks**

20	20	20	20	20	20	20

  - What's a bad stream of frees and then allocates?
  - Free every other chunk, then alloc 21 bytes
- **Next: two allocators (best fit, first fit) that, in practice, work pretty well**
  - "pretty well" = ~20% fragmentation under many workloads

Best Fit

- **Strategy: minimize fragmentation by allocating space from block that leaves smallest fragment**
  - Data structure: heap is a list of free blocks, each has a header holding block size and a pointer to the next block



  - Code: Search freelist for block closest in size to the request. (Exact match is ideal)
  - During free return free block, and (usually) coalesce adjacent blocks
- **Potential problem: Sawdust**
  - Remainder so small that over time left with “sawdust” everywhere
  - Fortunately not a problem in practice

Best Fit Gone Wrong

- **Simple bad case: allocate n, m $(n < m)$ in alternating orders, free all the ns, then try to allocate an $n + 1$**
- **Example: start with 99 bytes of memory**
  - alloc 19, 21, 19, 21, 19

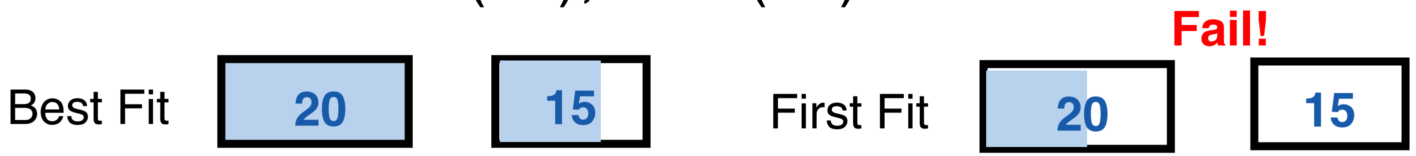
  - free 19, 19, 19:

  - alloc 20? Fails! (wasted space = 57 bytes)
- **However, doesn't seem to happen in practice**

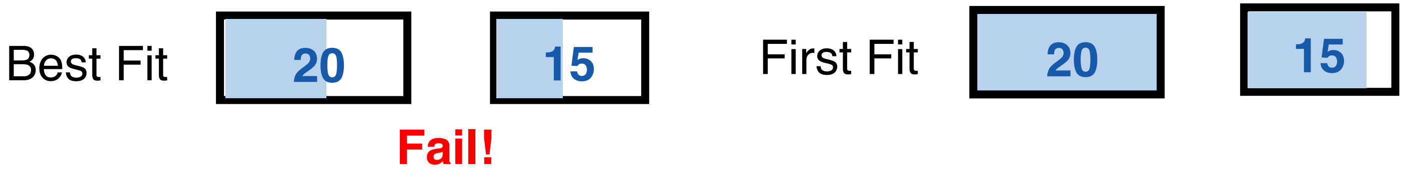
First Fit

- **Strategy: pick the first block that fits**
  - Data structure: free list, sorted LIFO, FIFO, or by address
  - Code: scan list, take the first one

- **Suppose memory has free blocks:** 20 15
  - Workload 1: alloc(10), alloc(20)



  - Workload 2: alloc(8), alloc(12), alloc(12)



First Fit

- **LIFO: put free object on front of list.**
  - Simple, but causes higher fragmentation
  - Potentially good for cache locality
- **Address sort: order free blocks by address**
  - Makes coalescing easy (just check if next block is free)
  - Also preserves empty/idle space (locality good when paging)
- **FIFO: put free object at end of list**
  - Gives similar fragmentation as address sort, but unclear why

Subtle Pathology: LIFO FF

- **Storage management example of subtle impact of simple decisions**
- **LIFO first fit seems good:**
  - Put object on front of list (cheap), hope same size used again (cheap + good locality)
- **But, has big problems for simple allocation patterns:**
  - E.g., repeatedly intermix short-lived $2n$-byte allocations, with long-lived $(n + 1)$-byte allocations
  - Each time large object freed, a small chunk will be quickly taken, leaving useless fragment. Pathological fragmentation

First Fit: Nuances

- **First fit sorted by address order, in practice**
  - Blocks at front preferentially split, ones at back only split when no larger one found before them
  - Result? Seems to roughly sort free list by size
  - So? Makes first fit operationally similar to best fit: a first fit of a sorted list = best fit!
- **Problem: sawdust at beginning of the list**
  - Sorting of list forces a large requests to skip over many small blocks. Need to use a scalable heap organization

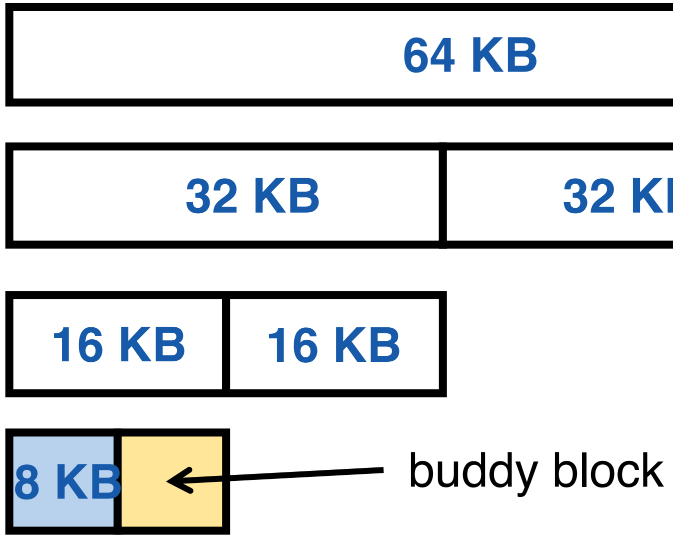
Some Other Ideas

- **Worst-fit:**
  - Strategy: fight against sawdust by splitting blocks to maximize leftover size
  - In real life seems to ensure that no large blocks around
- **Next fit:**
  - Strategy: use first fit, but remember where we found the last thing and start searching from there
  - Seems like a good idea, but tends to break down entire list
- **Buddy systems:**
  - Round up allocations to power of 2 to make management faster

Buddy Allocator Motivation

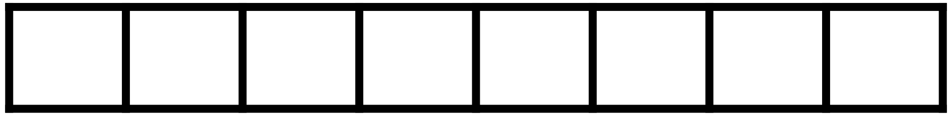
- **Allocation requests: frequently 2^n**
  - E.g., allocation physical pages in Linux
  - Generic allocation strategies: overly generic
- **Fast search (allocate) and merge (free)**
  - Avoid iterating through free list
- **Avoid external fragmentation for req of 2^n**
- **Keep physical pages contiguous**
- **Used by Linux, FreeBSD**

Buddy Allocation



- **Recursively divide larger blocks until reach suitable block**
  - Big enough to fit but if further splitting would be too small
- **Insert “buddy” blocks into free lists**
- **Upon free, recursively coalesce block with buddy if buddy free**

Buddy Allocation Example

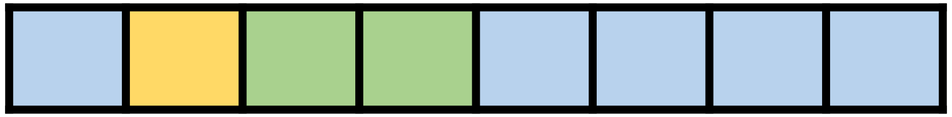


freelist[3] = {0}

`p1 = alloc(2^0)`

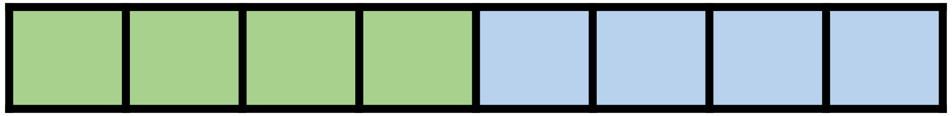
freelist[0] = {1}, freelist[1] = {2}, freelist[2] = {4}

`p2 = alloc(2^2)`



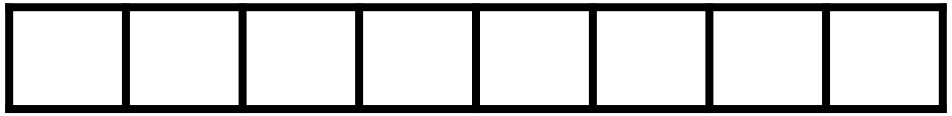
freelist[0] = {1}, freelist[1] = {2}

`free(p1)`



freelist[2] = {0}

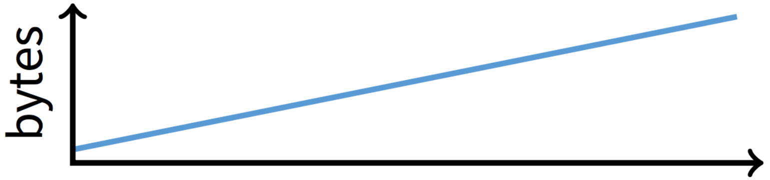
`free(p2)`



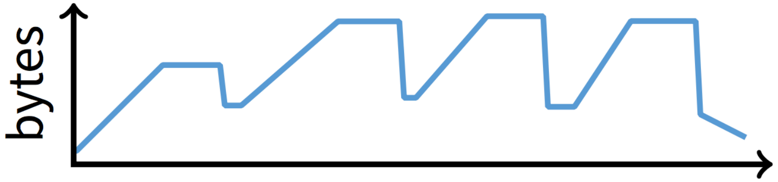
freelist[3] = {0}

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

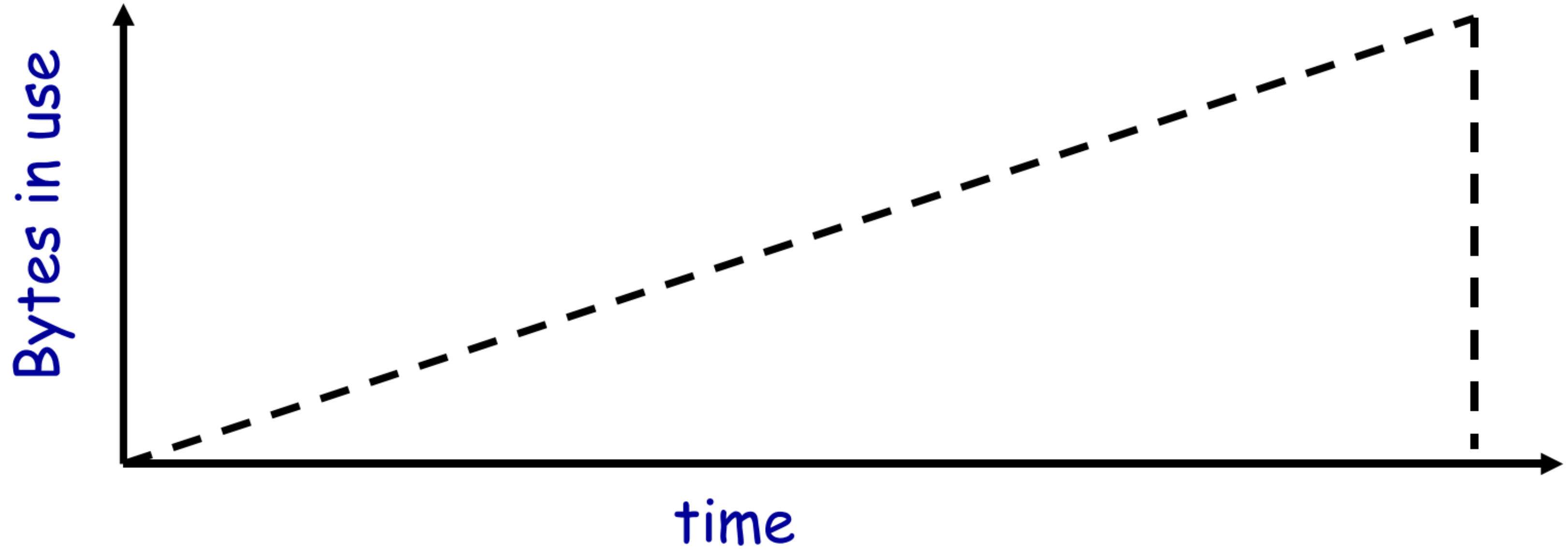


  - *Peaks*: allocate many objects, use briefly, then free all



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

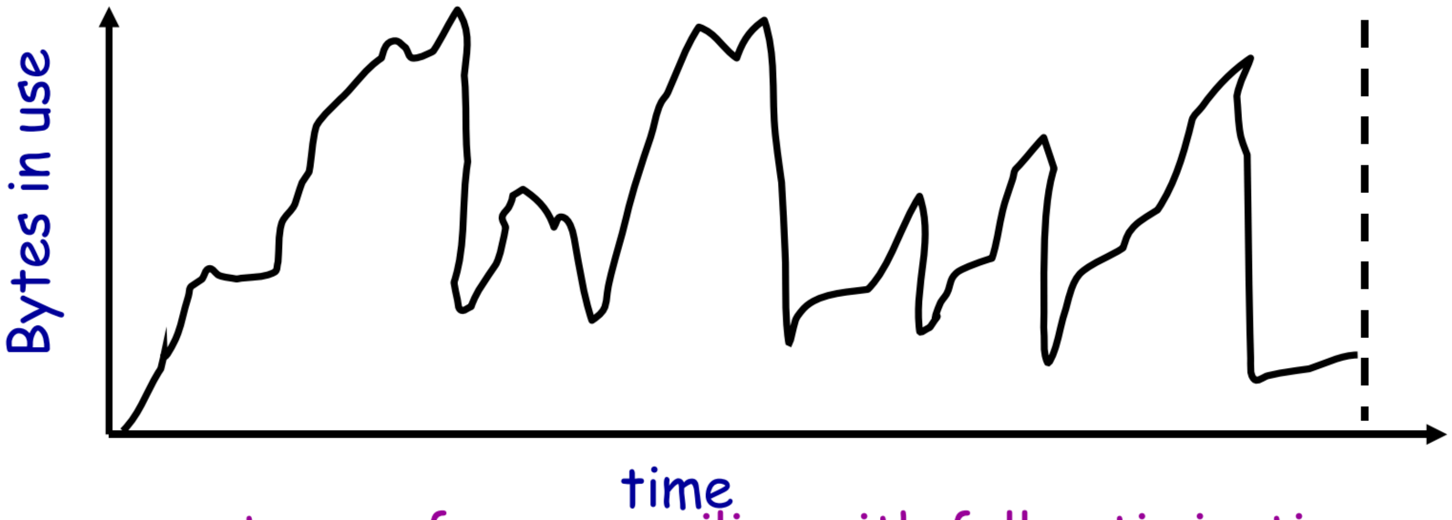
Pattern 1: ramps



trace from an LRU simulator

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

Pattern 2: Peaks



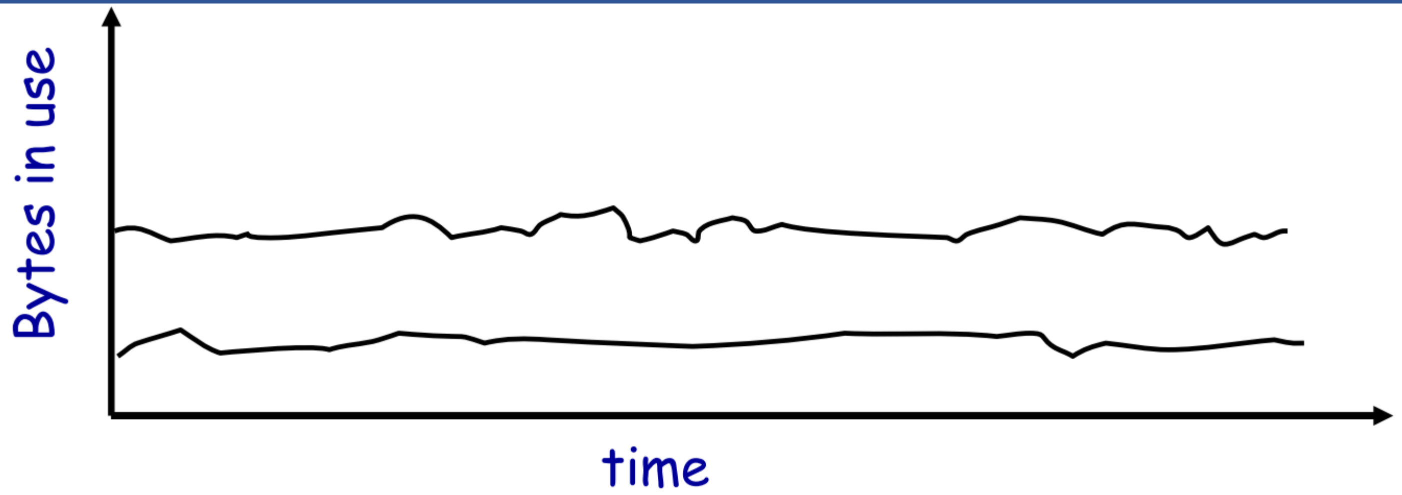
trace of gcc compiling with full optimization

- **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: allocate a lot, then free everything**
  - Change allocation interface: allocate as before, but only support free of everything all at once
  - 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: Plateaus



trace of perl running a string processing script

- **Plateaus: allocate many objects, use for a long time**
  - What happens if overlap with peak or different plateau?

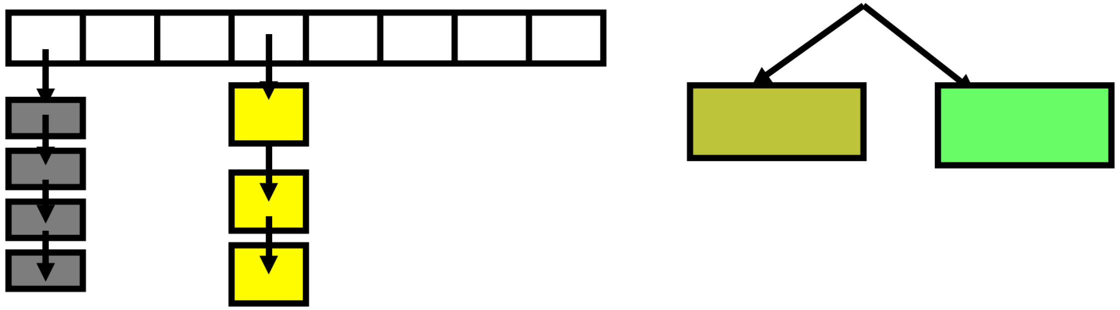
Slab Allocation

- **Kernel allocates many instances of same structures**
  - E.g., a 1.7 KB `task_struct` for every process on system
- **Often want contiguous physical memory (for DMA)**
- **Slab allocation optimizes for this case:**
  - A slab is multiple pages of contiguous physical memory
  - A cache contains one or more slabs
  - Each cache stores only one kind of object (fixed size)
- **Each slab is full, empty, or partial**

Slab Allocation

- **E.g., need new `task_struct`?**
  - Look in the `task_struct` cache
  - If there is a partial slab, pick free `task_struct` in that
  - Else, use empty, or may need to allocate new slab for cache
- **Free memory management: bitmap**
  - Allocate: set bit and return slot, Free: clear bit
- **Advantages: speed, and no internal fragmentation**
- **Used in FreeBSD and Linux, implemented on top of buddy page allocator**

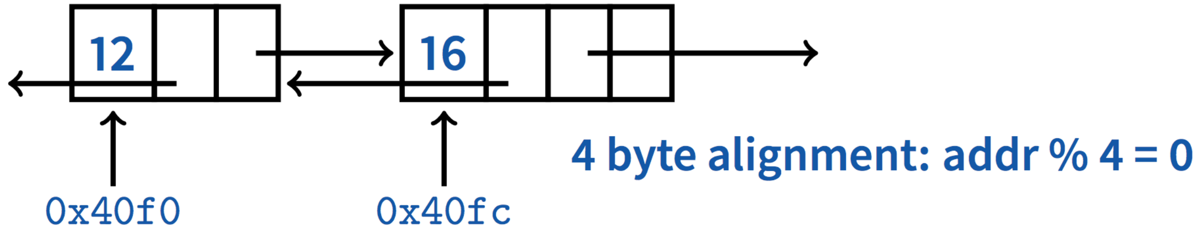
Simple, Fast Segregated Free Lists



- **Array of free lists for small sizes, tree for larger**
  - Place blocks of same size on same page
  - Have count of allocated blocks: if goes to zero, can return page
- **Pro: segregate sizes, no size tag, fast small alloc**
- **Con: worst case waste: 1 page per size even w/o free, After pessimal free: waste 1 page per object**
- **TCMalloc [Ghemawat] is a well-documented malloc like this**

Typical Space Overheads

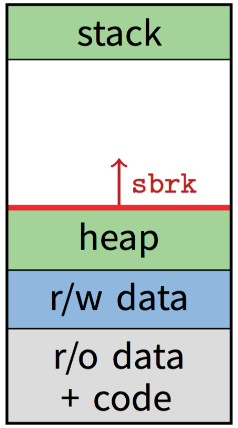
- **Free list bookkeeping and alignment determine minimum allocatable size:**
- **If not implicit in page, must store size of block**
- **Must store pointers to next and previous freelist element**



- **Allocator doesn't know types**
  - Must align memory to conservative boundary

Getting More Space from OS

- **On Unix, can use sbrk**
  - E.g., to activate a new zero-filled page:




```
/* add nbytes of valid virtual address space */
void *get_free_space(size_t nbytes) {
  void *p = sbrk(nbytes);
  if (!p)
    error("virtual memory exhausted");
  return p;
}
```

- **For large allocations, sbrk a bad idea**
  - May want to give memory back to OS
  - Can't with sbrk unless big chunk last thing allocated
  - So allocate large chunk using `mmap`'s MAP_ANON

Next Time…

- **Read Chapter 36, 37**