

CSL373: Lecture 14

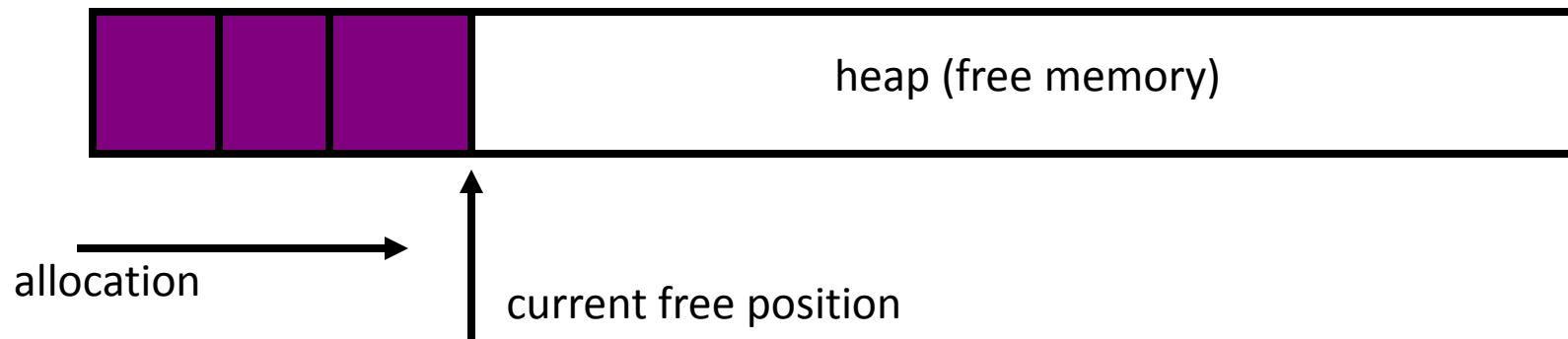
User level memory management

Today: dynamic memory allocation

- Almost every useful program uses dynamic allocation:
 - 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
- Today: how to implement, what's hard.
- 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.

What's the goal? And why is it hard?

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

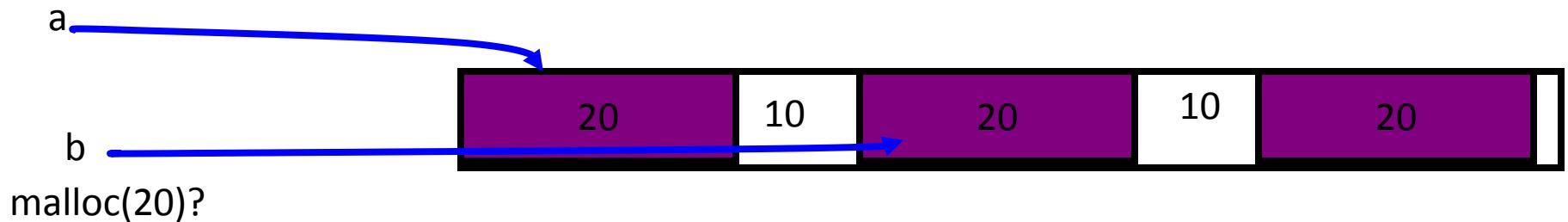


- Problem: free creates holes ("fragmentation") Result? Lots of free space but cannot satisfy request!



More abstractly

- 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
 - change user ptrs = (bad) placement decisions permanent



- 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 causes
 - Different lifetimes: if adjacent objects die at different times, then fragmentation:



- if they die at the same time, then no fragmentation:



- Different sizes: if all requests the same size, then no fragmentation (paging artificially creates this)



The important decisions for fragmentation

- 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)
- Splitting free blocks to satisfy smaller requests
 - fights internal fragmentation
 - freedom: can chose any larger block to split
 - one way: chose block with smallest remainder (best fit)
- Coalescing free blocks to yield larger blocks
 - freedom: when coalescing done (deferring can be good)
 - fights external fragmentation



Impossible to “solve” fragmentation

- If you read allocation papers or books to find the best allocator(!?!) it can be frustrating:
 - 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.
- What is bad?
 - Good allocator: $M * \log(n)$ where M = bytes of live data and n = ratio between smallest and largest sizes.
 - Bad allocator: $M * n$

Pathological examples

- Given allocation of 7 20-byte chunks



- What's a bad stream of frees and then allocates?
- Given 100 bytes of free space

A single horizontal bar labeled '100' in the center. The bar is enclosed in a thick black border.

- What's a really bad combination of placement decisions and malloc & frees?
- 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 pointers to next



- Code: Search freelist for block closest in size to the request.
(Exact match is ideal)
- During free (usually) coalesce adjacent blocks
- 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 ($m < n$) in alternating orders, free all the m 's, then try to allocate an $m+1$.
- Example: start with 100 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

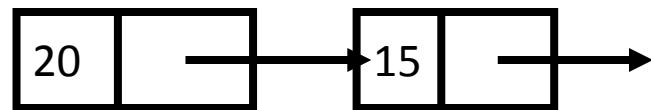
- Strategy: pick the first block that fits
 - Data structure: free list, sorted lifo, fifo, or by address
 - Code: scan list, take the first one.
- LIFO: put free object on front of list.
 - Simple, but causes higher fragmentation
- Address sort: order free blocks by address.
 - Makes coalescing easy (just check if next block is free)
 - Also preserves empty space (good)
- FIFO: put free object at end of list.
 - Gives ~ fragmentation as address sort, but unclear why

An example 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:
 - repeatedly intermix short-lived large allocations, with long-lived small allocations.
 - Each time large object freed, a small chunk will be quickly taken. Pathological fragmentation.

First fit: Nuances

- First fit + 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
- When better than best fit?
 - Suppose memory has free blocks:
 - Suppose allocation ops are 10 then 20
 - Suppose allocation ops are 8, 12, then 12



The strange parallels of first and best fit

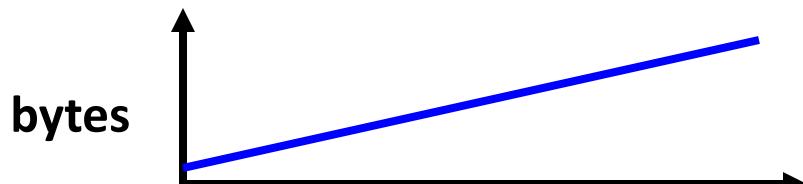
- Both seem to perform roughly equivalently
- In fact the placement decisions of both are roughly identical under both randomized and real workloads!
 - Pretty strange since they seem pretty different
- Possible explanations:
 - first fit ~ best fit because over time its free list becomes sorted by size: the beginning of the free list accumulates small objects and so fits tend to be close to best
 - both have implicit “open space hueristic” try not to cut into large open spaces: large blocks at end only used until have to be (e.g., first fit: skips over all smaller blocks)

Some worse ideas

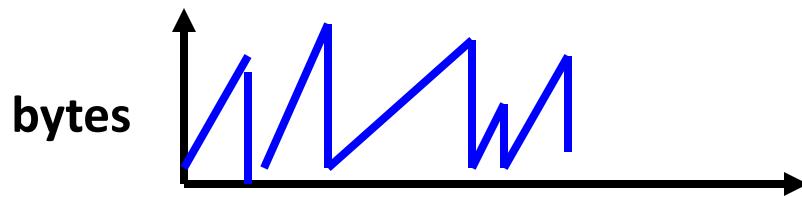
- Worst-fit:
 - strategy: fight against sawdust by splitting blocks to maximize leftover size
 - in real life seems to ensure that no large blocks around
- Next fit:
 - strategy: use first fit, but remember where we found the last thing and start searching from there.
 - Seems like a good idea, but tends to break down entire list
- Buddy systems:
 - round up allocations to make management faster
 - result? heavy internal fragmentation

Known patterns of real programs

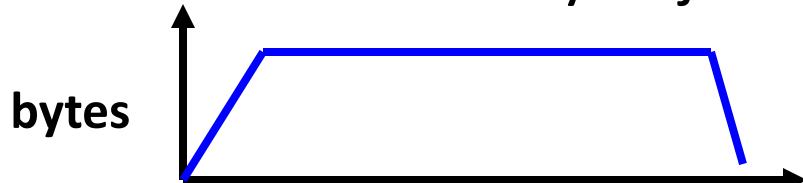
- So far we've treated programs as black boxes.
- Most real programs exhibit 1 or 2 (or all 3) of the following patterns of alloc/dealloc:
 - ramps: accumulate data monotonically over time



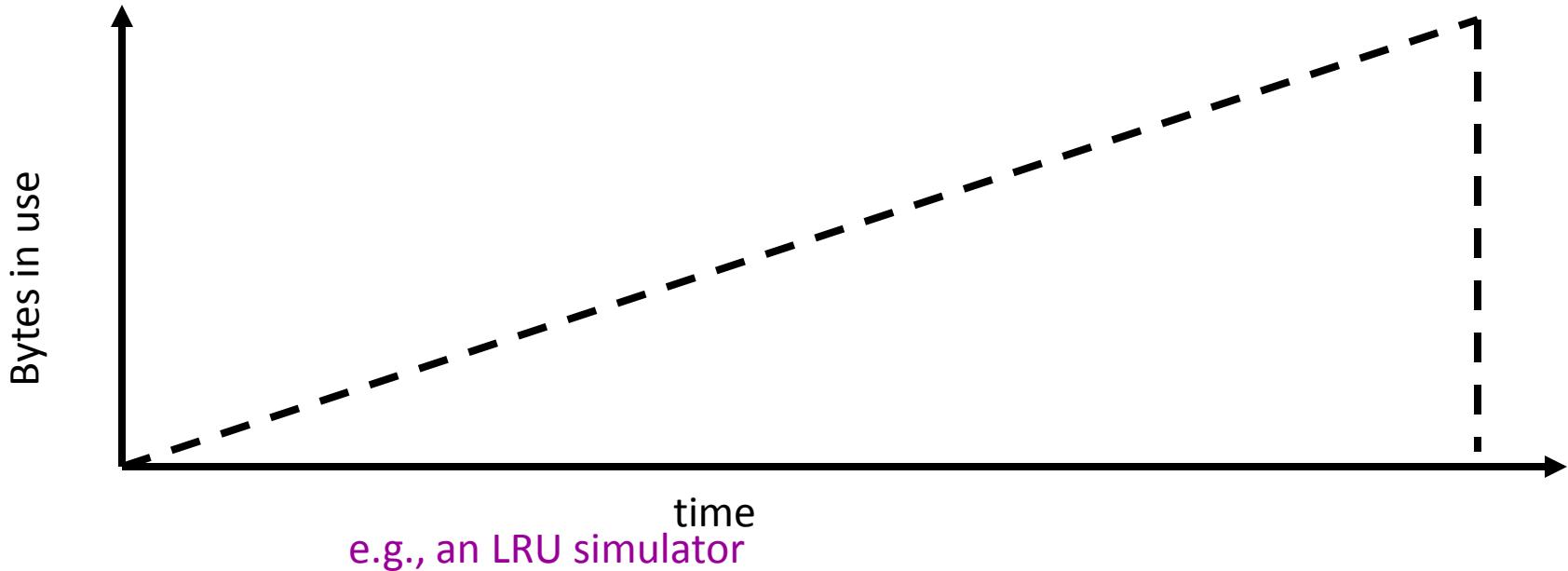
- peaks: allocate many objects, use briefly, then free all



- plateaus: allocate many objects, use for a long time



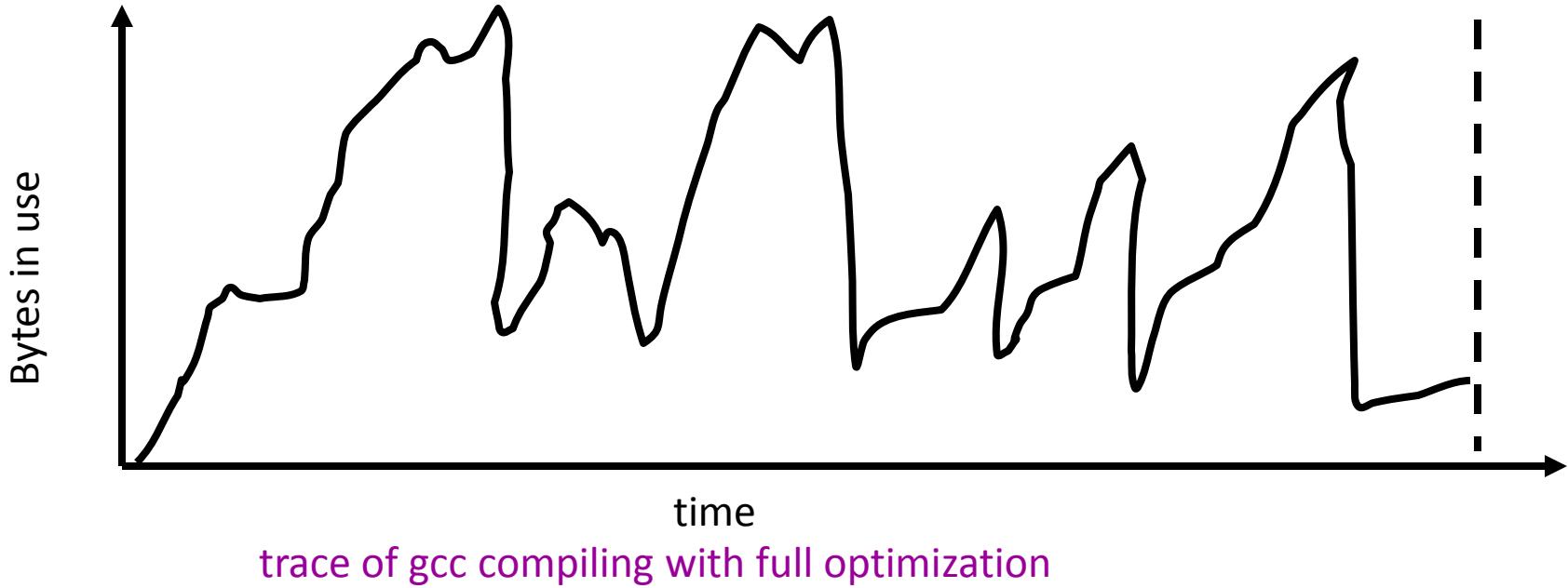
Pattern 1: ramps



e.g., an LRU simulator

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

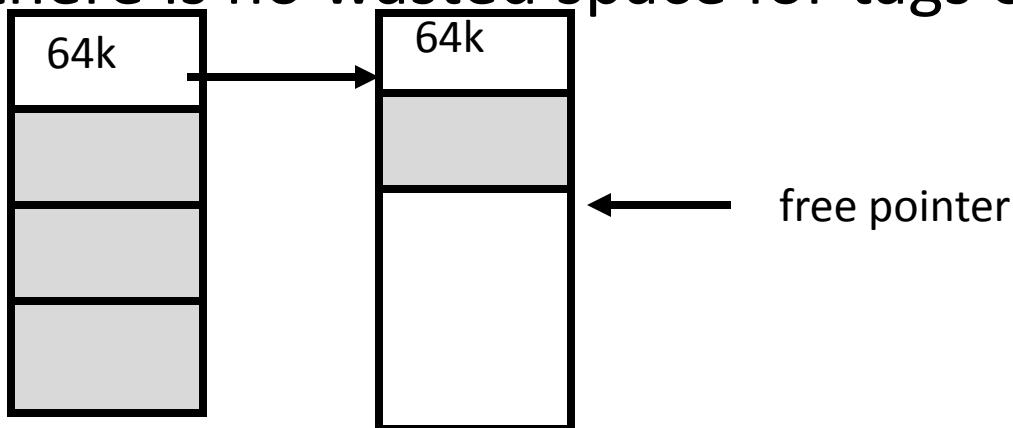
Pattern 2: peaks



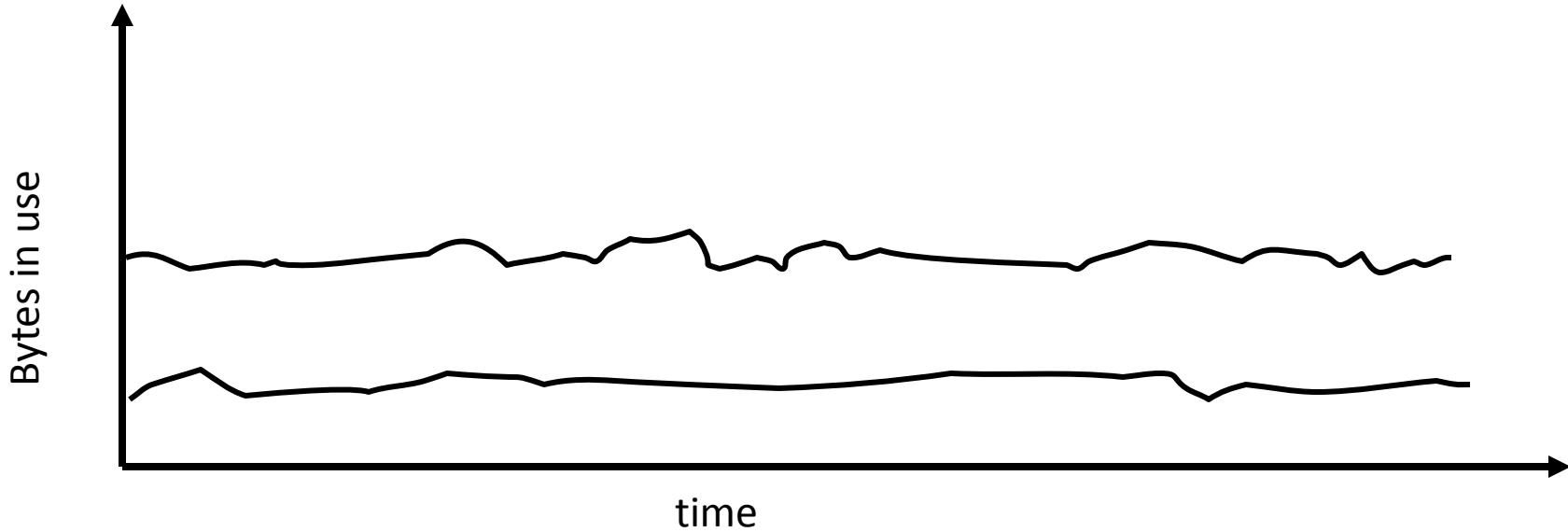
- Peaks: allocate many objects, use briefly, then free all
 - Fragmentation a real danger.
 - Interleave peak & ramp? Interleave two different peaks?
 - What happens if peak allocated from contiguous memory?

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 procedure call (by compiler people)
- arena = a linked list of large chunks of memory.
 - Advantages: alloc is a pointer increment, free is “free”, & there is no wasted space for tags or list pointers.



Pattern 3: Plateaus

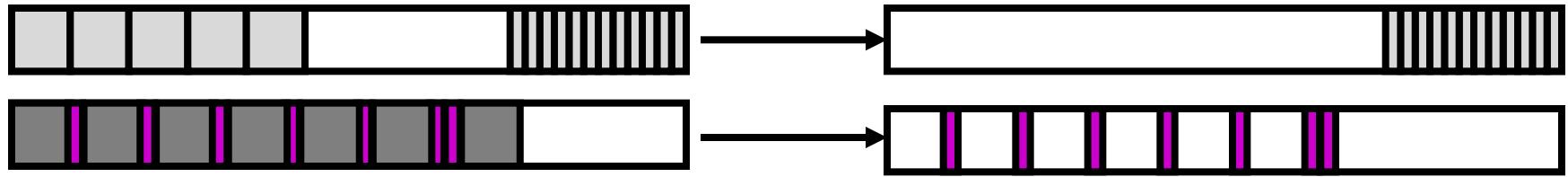


trace of perl running a string processing script

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

Some observations to fight fragmentation

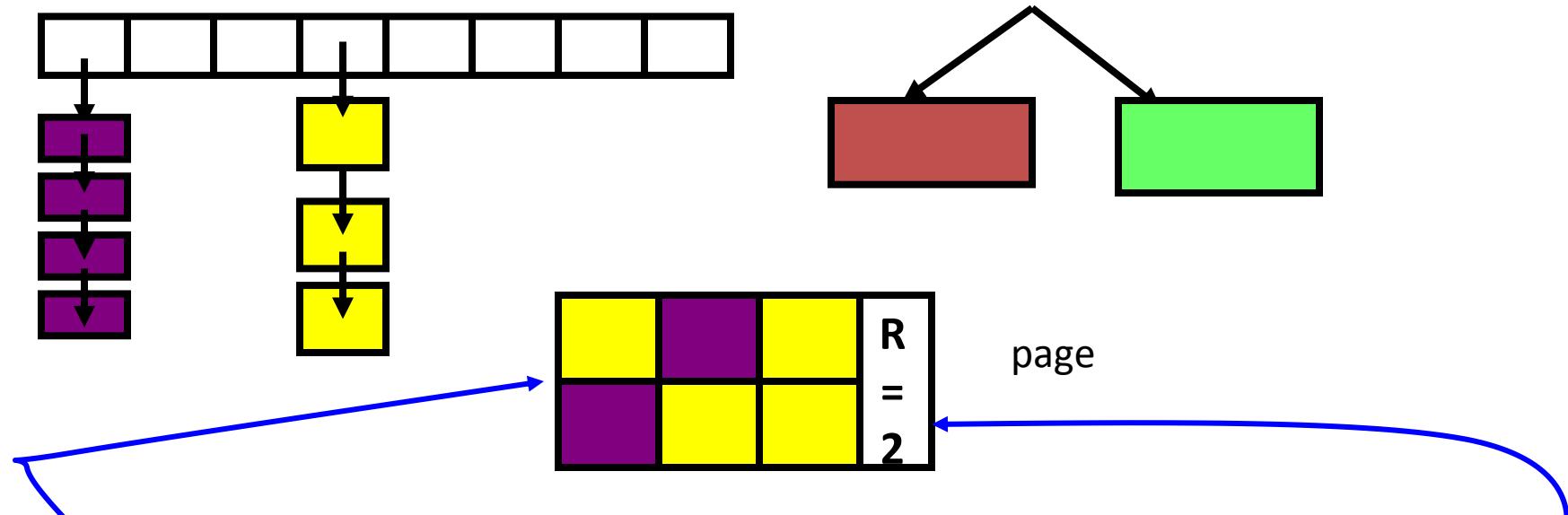
- Segregation = reduced fragmentation:
 - Allocated at same time ~ freed at same time
 - Different type ~ freed at different time



- Implementation observations:
 - Programs allocate small number of different sizes
 - Fragmentation at peak use more important than at low
 - Most allocations small (< 10 words)
 - Work done with allocated memory increases with size.
 - Implications?

Simple, fast segregated free lists

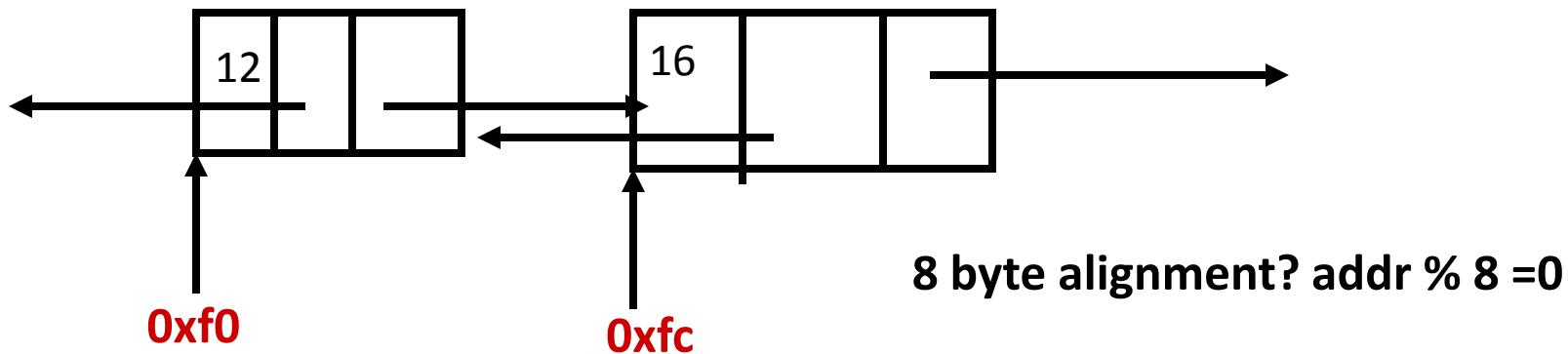
- Array of with free list to small sizes, tree for larger



- 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 + very fast small alloc
- Con: worst case waste: 1 page per size.

Typical space overheads

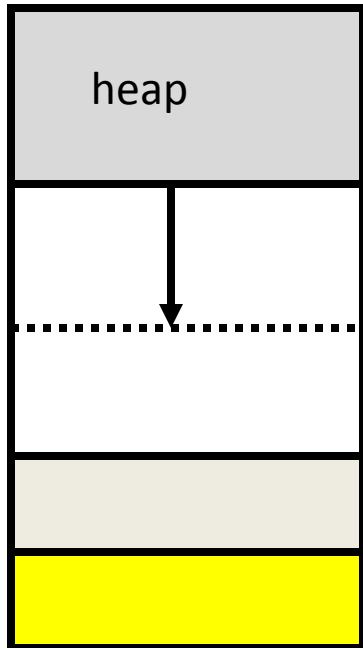
- Free list bookkeeping + alignment determine minimum allocatable size:
 - store size of block
 - pointers to next and previous freelist element



- Machine enforced overhead: alignment. Allocator doesn't know type. Must align memory to conservative boundary.
- Minimum allocation unit? Space overhead when allocated?

How do you actually get space?

- On Unix use sbrk to grow process's heap segment:

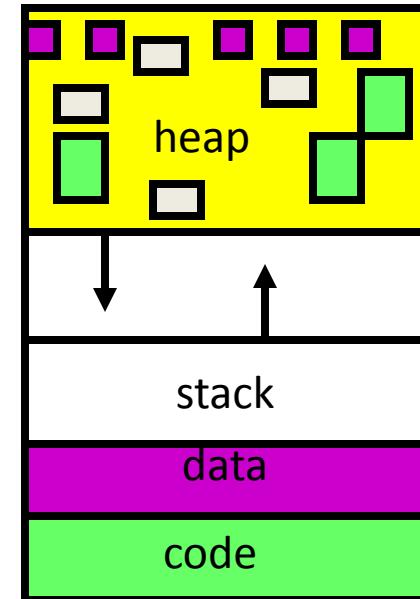


```
/* add nbytes of valid virtual address space */
void *get_free_space(unsigned nbytes) {
    void *p;
    if(!(p = sbrk(nbytes)))
        error("virtual memory exhausted");
    return p;
}
```

- Activates a zero-filled page sized chunk of virtual address space.
 - Remove from address space with sbrk(-nbytes)
 - This last block called “the wilderness”

Malloc versus OS memory management

- Relocation:
 - Virtual memory allows OS to relocate physical blocks (just update page table) as a result, it can compact memory.
 - User-level cannot. Placement decisions permanent
- Size and distribution:
 - OS: small number of large objects
 - malloc: huge number of small objs
 - internal fragmentation more important
 - speed of allocation very important
- Duplication of data structures
 - malloc memory management layered on top of VM
 - why can't they cooperate?



Fragmentation generalized

- Whenever we allocate, fragmentation is a problem
 - CPU, memory, disk blocks, ...
 - more general stmt: “the inability to use X that is free”
- Internal fragmentation:
 - How does malloc minimize internal fragmentation?
 - What corresponds to internal fragmentation of a process’s time quanta? How does scheduler minimize this?
 - In a book? How does the English language minimize? (and: Page size tradeoffs?)
- External frag = cannot satisfy allocation request
 - why is external fragmentation not a problem with money?

Reclamation: beyond free

- Automatic reclamation:
 - User-level: Anything manual can be done wrong: storage deallocation a major source of bugs
 - OS level: OS must manage reclamation of shared resources to prevent evil things.
- How?
 - Easy if only used in one place: when ptr dies, deallocate (example?)
 - Hard when shared: can't recycle until all sharers done



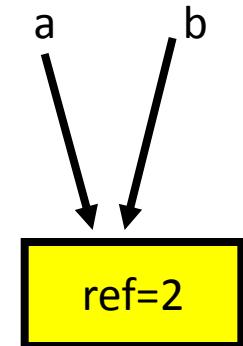
- sharing indicated by the presence of pointers to the data
- insight: no pointers to data = it's free!
- 2 schemes: ref counting; mark & sweep garbage collection

Reference counting

- Algorithm: counter pointers to object
 - each object has “ref count” of pointers to it
 - increment when pointer set to it
 - decremented when pointer killed

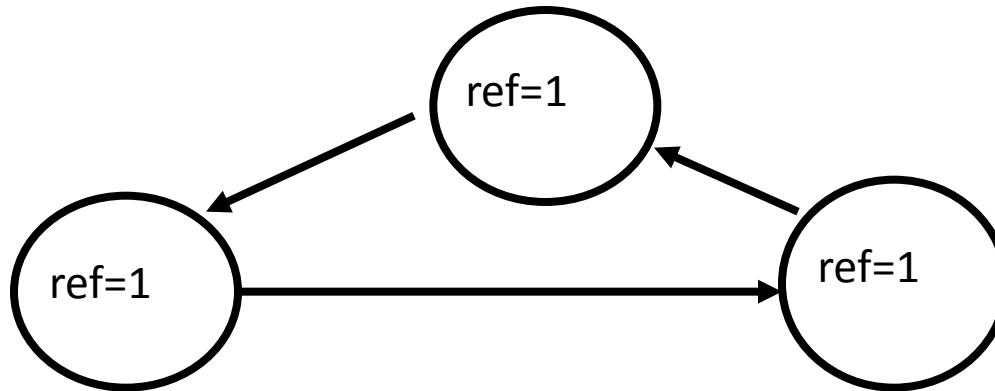
```
void foo(bar c) {  
    bar a, b;  
    a = c;    ←..... c->refcnt++;  
    b = a;    ←..... a->refcnt++;  
    a = 0;    ←..... a->refcnt--;  
    return;  ←..... b->refcnt--;  
}
```

- refcnt = 0? Free resource
- works fine for hierarchical data structures
 - file descriptors in Unix, pages, thread blocks



Problems

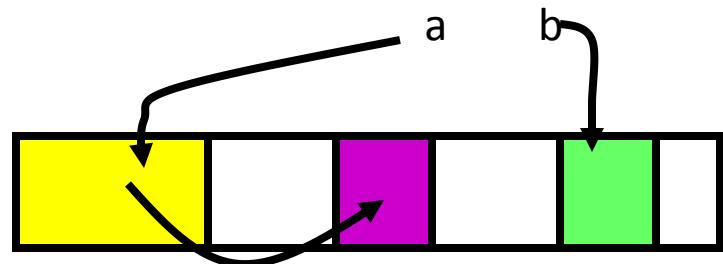
- Circular data structures always have `refcnt > 0`
 - if no external references = lost!



- Naïve: have to do on every object reference creation, deletion
 - Without compiler support, easy to forget decrement or increment. Nasty bug.

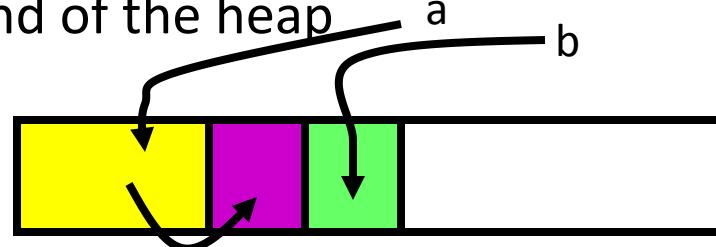
Mark & sweep garbage collection

- Algorithm: mark all reachable memory; rest is garbage
 - must find all “roots” - any global or stack variable that holds a pointer to an object.
 - Must find all pointers in objects



- pass 1: mark
 - mark memory pointed to by roots. Then recursively mark all objects these point to, ...
- pass 2: sweep

- go through all objects, free up those that aren't marked.
- Usually entails moving them to one end of the heap (compaction) and updating pointers



Some details

- Huge lever: Can update pointers
 - So can compact instead of running out of storage
 - Is fragmentation no longer an issue?
- Compiler support helps (to parse objects).
 - Java and Modula-3 support GC
- Can sort of do it without: “conservative gc”
 - at every allocation, record (address, size)
 - scan heap, data & stack for integers that would be legal pointer values and mark!

