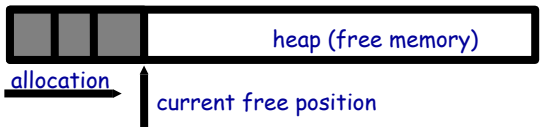
## Today: Dynamic Memory Allocation

- 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 50(!) years, memory management still poorly understood.**

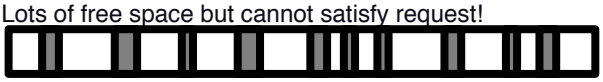
## What's the goal? And why is it hard?

Satisfy arbitrary set of allocations 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!



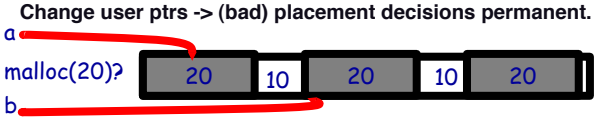
## More abstractly

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.**

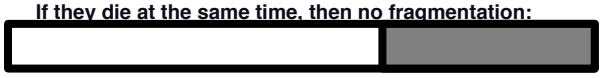
## What is fragmentation really?

"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

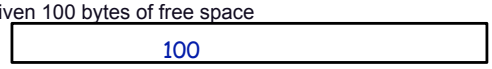
- If you read allocation papers or books to find the best allocator(!?!?!) it can be frustrating:
  - **All discussions revolve around tradeoffs.**
  - **The reason? There cannot be a best allocator.**
- Theoretical result:
  - **For any possible allocation algorithm, there exist streams of allocation 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




**What's a really bad combination of placement decisions, mallocs & frees?**

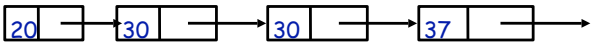
- Next: two allocators (best fit, first fit) that, in practice, work pretty well.

  **"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.**

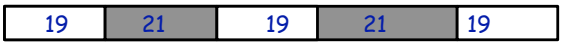
  **Fortunately not a problem in practice.**

## 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 (though the way real programs behave suggest it easily could

## 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**

## 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**

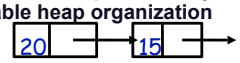
  **Result? Seems to roughly sort free list by size**

  **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 sizes are 10 then 20**

  **Suppose allocation sizes are 8, 12, then 12**

## The weird parallels of first and best fit

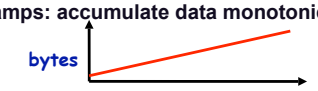
- Both seem to perform roughly equivalently
- In fact the placement decisions of both are roughly identical under both randomized and real workloads!
  - **No one knows why.**
  - **Pretty strange since they seem pretty different.**
- Possible explanations:
  - **First fit ~ best fit because over time 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 heuristic" 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

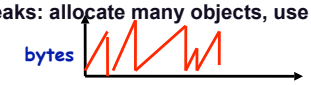
- Worst-fit:
  - **Strategy: fight against sawdust by splitting blocks to maximize leftover size**
  - **In real life seems to ensure that no large blocks around.**
- Next fit:
  - **Strategy: use first fit, but remember where we found the last thing and start searching from there.**
  - **Seems like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

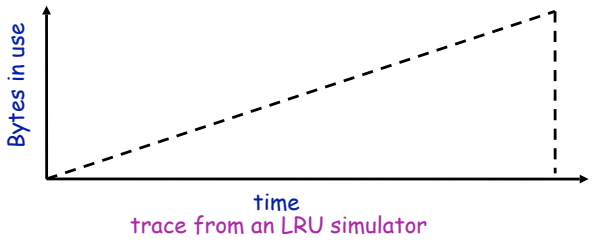



  - **peaks: allocate many objects, use briefly, then free all**




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

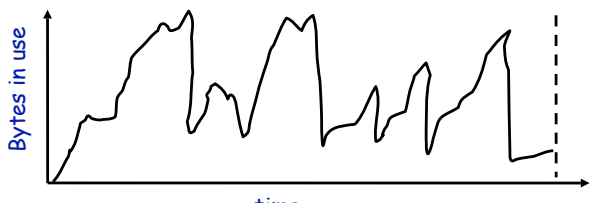

## Pattern 1: ramps




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

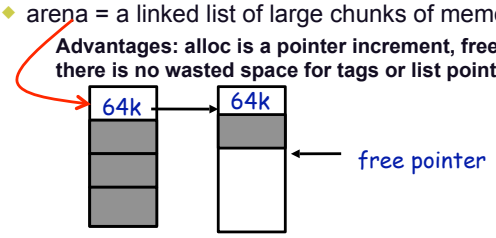



trace of gcc compiling with full optimization

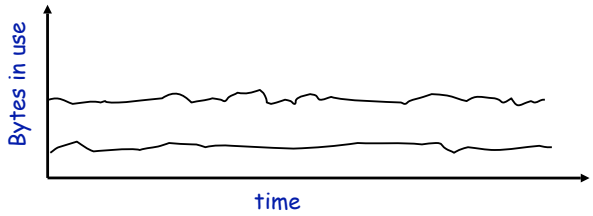
- Peaks: allocate many objects, use briefly, then free all
  - **Fragmentation a real danger.**
  - **Interleave peak & ramp? Interleave two different peaks?**
  - **What happens if peak allocated from contiguous memory?**

## 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

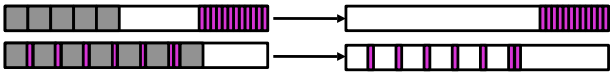



trace of perl running a string processing script

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

## Some observations to fight fragmentation

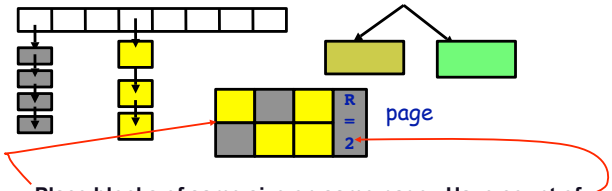
- Segregation = reduced fragmentation:
  - **Allocated at same time ~ freed at same time**
  - **Different type ~ freed at different time**



- 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**

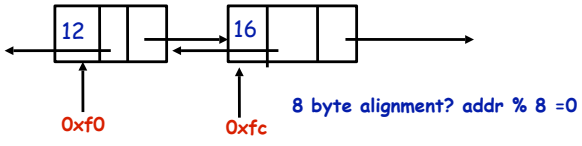
**Pro: segregate sizes + no size tag + very fast small alloc**
**Con: worst case waste: 1 page per size.**

## Example: slab allocation (Solaris, etc.)

- Kernel allocates many instances of same structure
  - **e.g. 1.7 KB `task_struct` for every process**
- 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**
- each slab is full, empty, or partial
- e.g. need new `task_struct`?
  - **look in `task_struct` cache**
  - **if partial slab, pick free `task_struct`**
  - **otherwise, use empty slab or allocate a new slab**
- Similar to previous: speed, no internal fragmentation

## Issue: Typical space overheads

- Free list bookkeeping + alignment determine minimum allocatable size:
  - **Store size of block.**
  - **Pointers to next and previous freelist element.**

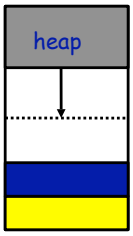



**Machine enforced overhead: alignment. Allocator doesn't know type. Must align memory to conservative boundary.**

**Minimum allocation unit? Space overhead when allocated?**

## How do you actually get space at user level?

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



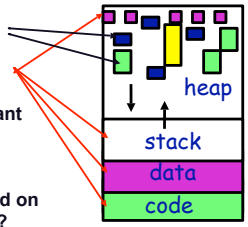

sbrk(4096)

```
/* 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)**

## `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 fragmentaion?**
  - **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?**

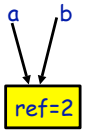
## Reclamation: beyond free

- Automatic reclamation:
  - **User-level: Anything manual can be done wrong: storage de-allocation a major source of bugs**
  - **OS level: OS must manage reclamation of shared resources to prevent bad (buggy, evil) things.**
- How?
  - **Easy if only used in one place: when ptr dies, deallocate**
  - **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.**

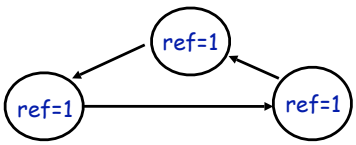



```
void foo(bar c) {
   bar a, b;
   a = c;  ..........c->refcnt++;
   b = a;  ..........a->refcnt++;
   a = 0;  ..........c->refcnt--;
   return; ..........b->refcnt--;
}
```

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

## Problems

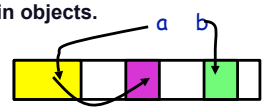
- Circular data structures always have refcnt > 0.
  - **if no external references = lost!**



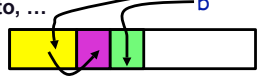

- Can do it without language support, but error-prone
  - **don't call `free()` - still have to call `release()`**
  - **override "=" (e.g. C++) -> "smart pointers"**
  - **but have to assign to null to release...**

## Mark & sweep garbage collection

- Algorithm: mark all reachable memory; rest is garbage
  - **Must find all "roots" - any global, stack variable, or register 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

- GC's 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, C#, JS, many dynamic lang's support GC**
  - **...but what about C/C++/etc.?**
- 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!**


## Garbage Collection vs. Reference Counting

- GC advantages
  - **automatic, hard to mess up**
  - **pointer update/copy is (potentially) chea**
  - **allocate sequentially? no free lists?**
- GC disadvantages
  - **must stop program, at least occasionally**
  - **weird, *unpredictable* latencies (!)**
- RC advantages
  - **No halts, predictable/bounded latency!**
- RC disadvantages
  - **pointer operations more expensive**
  - **cycles; programmer errors w/o language support**