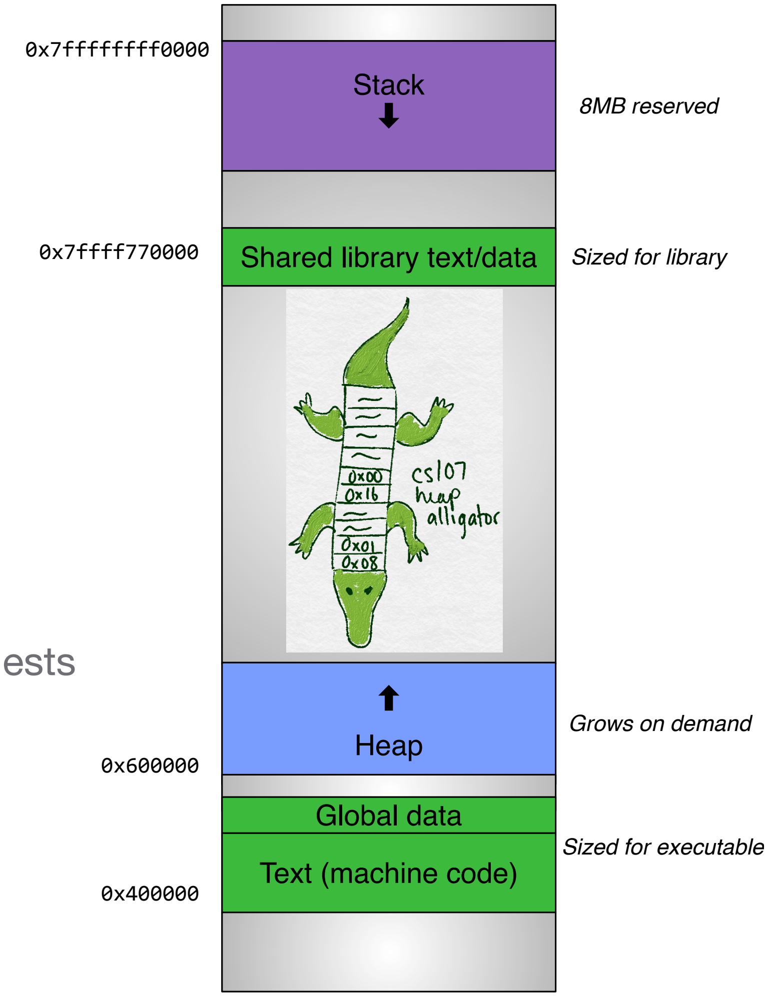
Program address space

- **What does the OS loader do?**
  - **Creates new process**
  - **Sets up address space/segments**
  - **Read executable file, load instructions, init global data**
    - Mapped from file into green segments
  - **Libraries loaded on demand**
  - **Set up stack**
    - Reserve stack segment, init %rsp, call main
  - **malloc written in C, will init self on use**
    - Asks OS for large memory region, parcels out to service requests



Thanks for the memory!

- **Global allocation**
  - **+/- Convenient, somewhat safe**
    - Automatic alloc/dealloc on program start/exit
    - Can access by name from anywhere
    - No encapsulation, hard to track use/dependencies
  - **- Size fixed at declaration, no option to resize**
  - **+/- Scope/lifetime is global/whole program**
    - One shared namespace, must manually avoid conflicts
- **Stack allocation**
  - **+ Efficient**
    - Fast to allocate/deallocate, ok to oversize
  - **+ Convenient, mostly safe**
    - Automatic alloc/dealloc on function entry/exit (can mistakenly return address of stack variable)
    - Reasonable type safety
  - **- Size fixed at declaration, no option to resize**
  - **+/- Scope/lifetime dictated by control flow**

Thanks for the memory (con't)

Heap allocation

+ Moderately efficient

Have to search for available space, update record-keeping

+ Very plentiful

Heap enlarges on demand to limits of address space

+ Versatile, under programmer control

Can precisely determine scope, lifetime

Can be resized

- Much opportunity for error

void* means effectively no type safety

Possible to allocate wrong size, use after free, double free, …

- Leaks

much less critical in grand scheme of things, but for long-running programs may be issue

Do we need all three options (globals/stack/heap)?

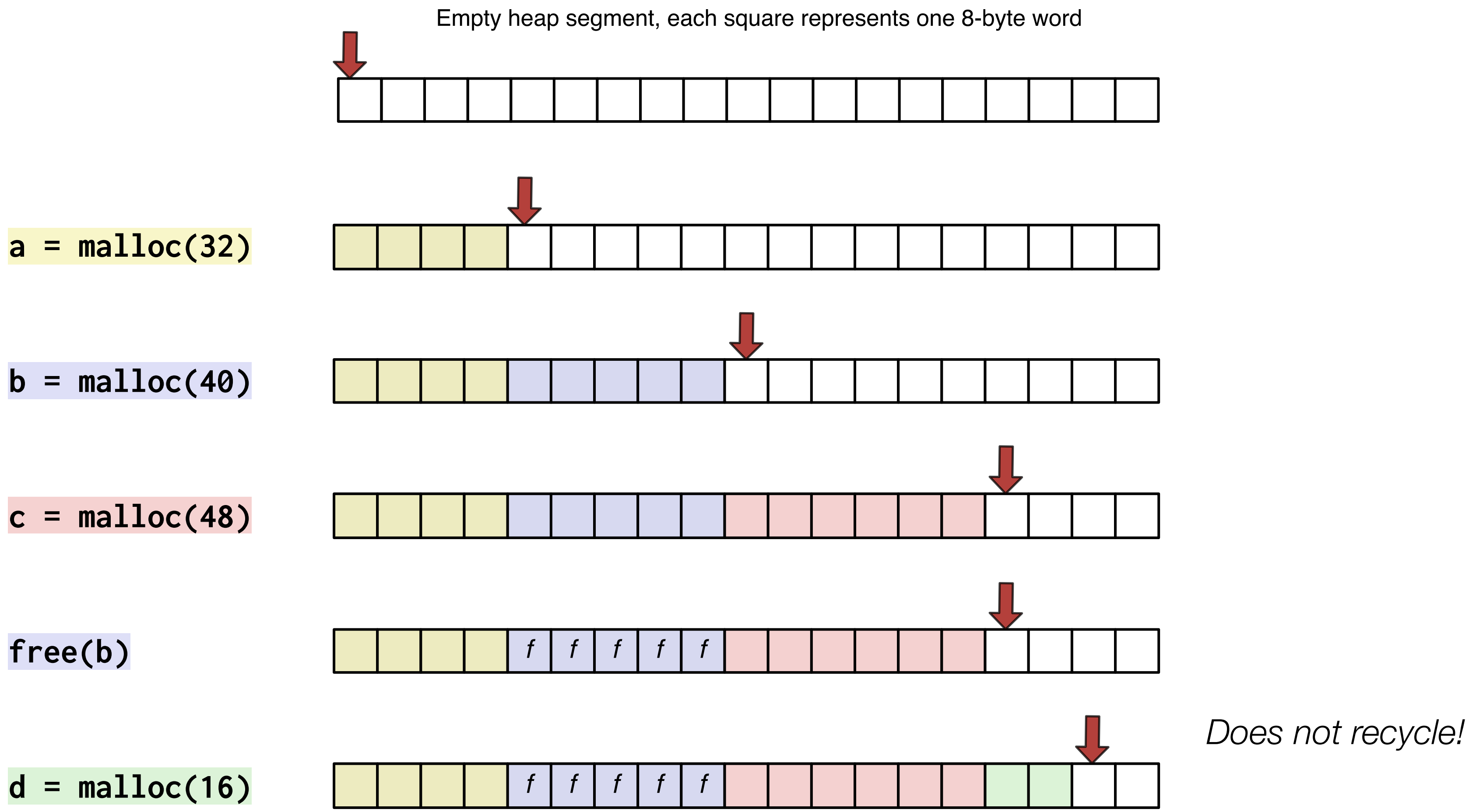
Heap allocator correctness

- **Service arbitrary sequence of malloc/realloc/free requests**
  - **Malloc returns pointer to memory block >= requested size (or NULL if cannot satisfy)**
  - **Payload contents unspecified (client can use calloc to zero if desired)**
  - **Client error results in undefined behavior (free non-malloc address, use freed memory, etc)**
- **Subject to constraints**
  - **Can't control number, size, lifetime of allocated blocks**
  - **Must respond immediately to each malloc request**
    - i.e., cannot reorder/buffer malloc requests
    - Can defer/ignore/reorder requests to free
  - **Must align blocks so they satisfy all alignment requirements**
    - Round up sizes (typically to multiple of 8 or 16)
  - **Allocated payload must be maintained as-is**
    - Cannot move allocated blocks, such as to compact/coalesce free, **why not?**
    - Can manipulate and modify memory not currently in use

Allocator goals

- **Non-negotiable: correctness**
  - Well-formed requests must be properly serviced
- **Highly desirable: performance**
  - Fast service of requests
    - Ideally constant-time, active/large heap should not bog down into linear behavior
  - Tight space utilization
    - Minimize fragmentation, allocated blocks grouped together, small overhead relative to payload
- **Possible tradeoffs:**
  - Ease of implementation/maintenance
    - Code often complex, be sure efforts are worthwhile (measure!)
  - Robust
    - Client errors generally blundered through, what is required to detect/report them? worth attempting?

Tracing a "bump" allocator



Code sketch: bump allocator

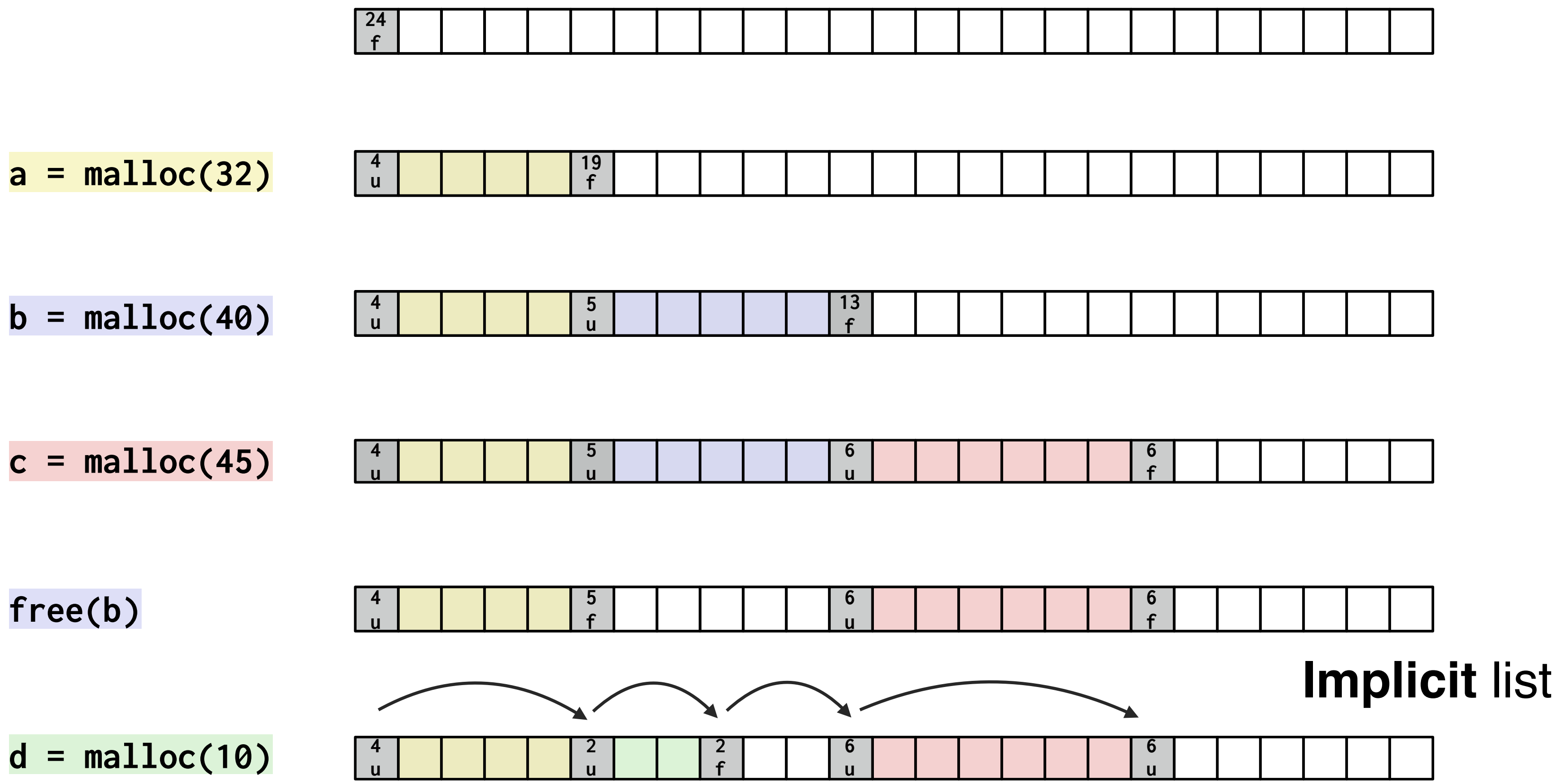
```
static void *segment_start;
static size_t segment_size, nused = 0;
// global variables segment_start/size track total heap segment


void *malloc(size_t nbytes)
{
    nbytes = roundup(nbytes, 16);
    if (nused + nbytes > segment_size) // not enough space
        return NULL;
    void *ptr = (char *)segment_start + nused;
    nused += nbytes;
    return ptr;
}

void free(void *ptr)
{
    // no-op! does not recycle used memory
}
```

Recycling

- **Must track block information to be able to recycle on free**
- **Separate housekeeping**
  - **Free/in-use information maintained in list/table**
    - Given address, how to look up information?
    - How to update to service malloc/free request?
    - How much overhead per-block?
  - **Seems reasonable approach, but not often used in practice**
    - Special-case allocators
    - Tools like Valgrind
- **Block header**
  - **Block information stored in memory that precedes payload**
    - Given address, how to look up information?
    - How to update to service malloc/free request?
    - How much overhead per-block?
  - **Most common approach in current use**

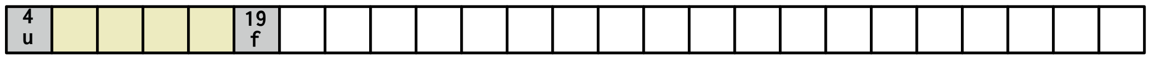
Tracing block header, recycling

Each square represents one 8-byte word, size in block header expressed in number of 8-byte words

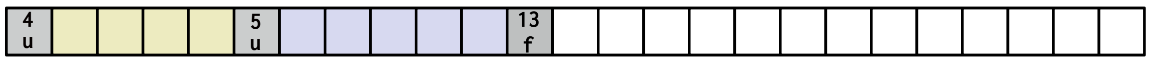


realloc can also recycle

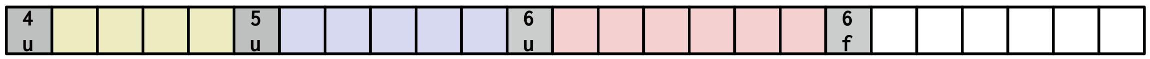
`a = malloc(32)`



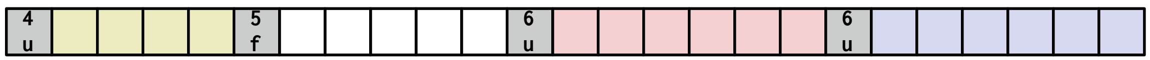
`b = malloc(40)`



`c = malloc(45)`



`b = realloc(b, 48)`

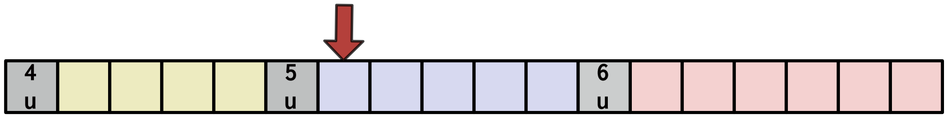


`a = realloc(a, 50)`

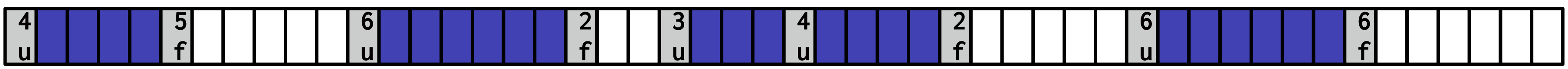
What is the advantage to an in-place realloc?

Code sketch: block header

```
#define FREE_BIT 1

struct header {
    unsigned long status; // bit mash size+free, free stored in lsb
};

struct header *ptr_to_header(void *ptr)
{
     return (struct header *)((char *)ptr - sizeof(struct header));
}


void free(void *ptr)
{
     struct header *hdr = ptr_to_header(ptr);
     hdr->status |= FREE_BIT;
}
```

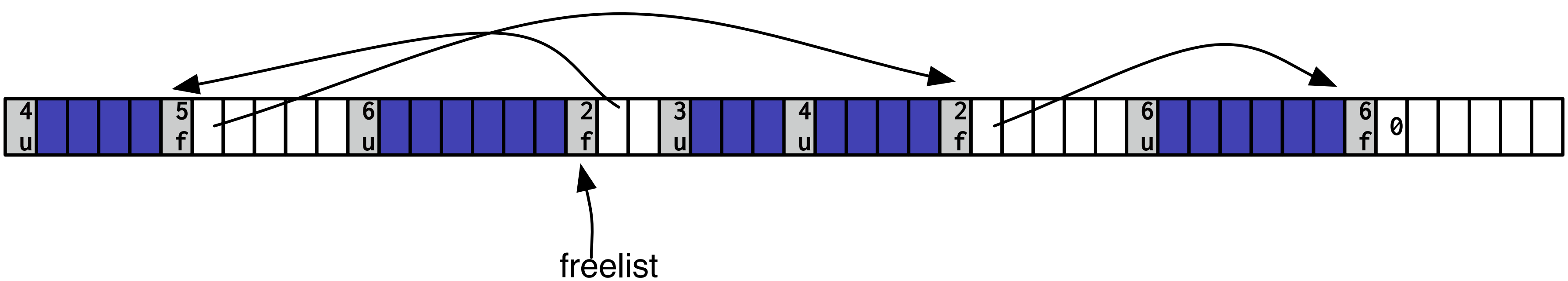


Adding an explicit free list



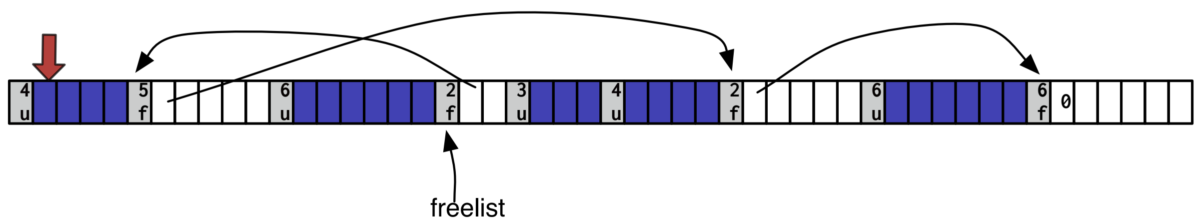
Traversing an implicit list bogs down as heap gets large/full

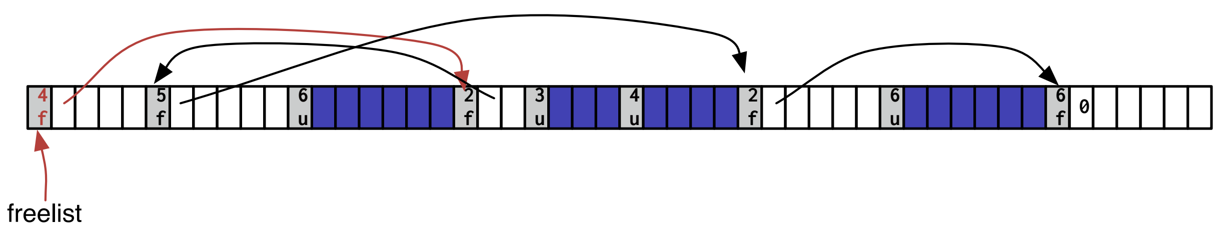
- Ideally, malloc only examines freed blocks
- Adding another data structure? hmmm…
- Idea: payload of freed blocks is available!



Code sketch: explicit list

```
struct header *freelist;

void free(void *ptr)
{
     struct header *hdr = ptr_to_header(ptr);
     hdr->status |= FREE_BIT;
     *(struct header **)ptr = freelist;
     freelist = hdr;
}
```





Managing free list

- **Implicit list**
  - Size in each block header allows traverse from block to block
  - Search visits all blocks to find free ones, becomes slow as heap fills up
- **Explicit list**
  - Chain free blocks into linked list
    - Why allowed/desirable to use the payload to store the links?
  - Search looks only free blocks!
- **Can be sorted or segregated (by size)**
  - Quickly access appropriate blocks for requested size — why valuable?
  - If sorted, what data structures to use — needs to quick to update…
  - If segregated, how many/what size classes to use?
- **Tradeoffs**
  - Additional overhead (minimum payload size)
  - More complex code to maintain/update

Policy decisions

- **Placement policy**
  - First-fit, next-fit, best-fit
  - Trades throughput for utilization
- **Splitting policy**
  - When to leave excess and when to split into separate node
    - (In my grandmother's attic: "Pieces of string too short to save"…)
- **Coalescing policy**
  - Immediate coalescing: coalesce each time free() is called
  - Deferred coalescing: try to improve performance of free() by deferring coalescing until needed. Examples:
    - Coalesce as you scan the free list for malloc()
    - Coalesce when the amount of external fragmentation reaches some threshold
  - Tension between split and coalesce — may do/undo for no benefit

How to make operations fast?

- malloc is generally about search
  - Make it faster by more quickly identifying which block to use
  - Examine fewer blocks
  - Be less picky about which block to use
- free is mostly about update
  - Ideal data structure can be modified in constant-time
  - Possibly postpone work till clearly needed (immediate vs deferred coalesce)
- realloc generally rides on malloc/free, resize in place if possible!
  - Big win if avoid copy payload data
    - What is necessary to allow resize in place? Is it worth it to anticipate that? How prominent is realloc in mix of operations?
- Heap allocator coding requires "scrappy" mindset
  - Pare down to tens of instructions per-request, every instruction counts!