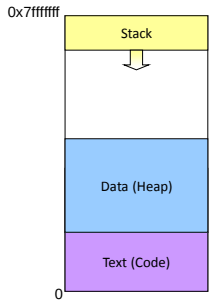
Dynamic Memory

Process's Address Space



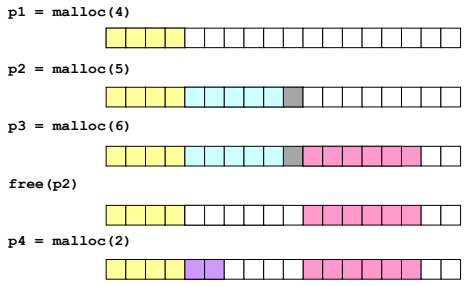
Backing the Heap

- When a process starts the heap is empty
  - The process is responsible for requesting memory from the OS
    - System Call: `void * sbrk(int increment)`
      - Basically: The process asks for an increase of a certain size
  - The OS provides a large contiguous memory space, which the process manages itself

Dynamic memory allocation

- Two basic memory allocator types:
  - Explicit: application allocates and frees space
    - E.g., malloc and free in C
  - Implicit: application allocates, but does not free space
    - E.g. garbage collection in Java, ML or Lisp
- Allocation
  - In both cases the memory allocator provides an abstraction of memory as a set of blocks
  - Doles out free memory **blocks** to application

Allocation examples

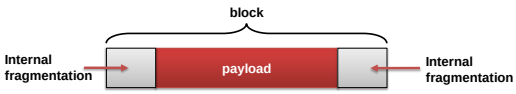


Goals of good malloc/free

- Primary
  - Good time performance for malloc and free
    - Ideally should take constant time (not always possible)
    - Should certainly not take linear time in the number of blocks
  - Good space utilization
    - User allocated structures should be large fraction of the heap
    - Want to minimize "fragmentation"
- Some Others
  - Good locality properties
    - Structures allocated close in time should be close in space
    - "Similar" objects should be allocated close in space
  - Robust
    - Can check that free(p1) is on a valid allocated object p1
    - Can check that memory references are to allocated space

Internal fragmentation

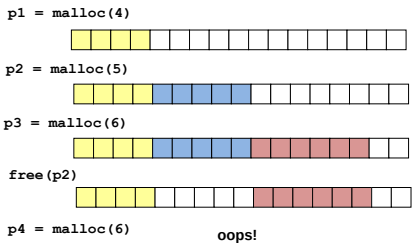
- Fragmentation causes poor memory utilization
  - Comes in two forms: internal and external fragmentation
- Internal fragmentation
  - The difference between the block size and the payload size



  - Due to:
    - overhead of maintaining heap data structures
    - padding for alignment purposes
    - explicit policy decisions (e.g., not to split the block).
  - Depends only on the pattern of previous requests, so easy to measure

External fragmentation

Occurs when there is enough aggregate heap memory, but no single free block is large enough



Depends on the pattern of *future* requests, so it's difficult to measure

What to blocks to allocate?

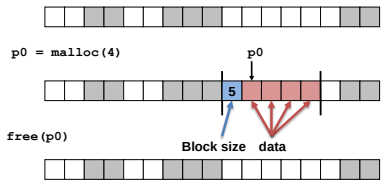
- **First fit:**
  - Search list from beginning, select first free block that fits
  - Can take linear time in num. of blocks (allocated and free)
  - In practice it can cause "splinters" at beginning of list
- **Next fit:**
  - Like first-fit, but start from end of previous search
  - Research suggests that fragmentation is worse
- **Best fit:**
  - Search the list, select free block with closest size that fits
  - Keeps fragments small --- usually helps fragmentation
  - Will typically run slower than first-fit

Implementation issues

- How do we know how much memory to free just given a pointer?
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
- How do we pick a block to use for allocation – many might fit?
- How do we reinsert freed block?

Knowing how much to free

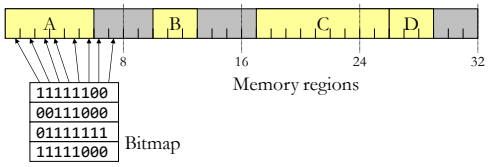
- Standard method
  - Keep length of a block in the word preceding the block
    - This word is often called the header field or header
  - Requires an extra word for every allocated block



What is the first thing free has to do?

Keeping track of free blocks (The Easy Way)

Bitmap

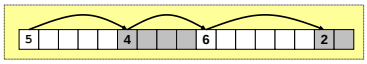


Pros:
- **EASY**

Cons:
- Takes up a lot of space

Keeping track of free blocks

- **Implicit list** using lengths -- links all blocks

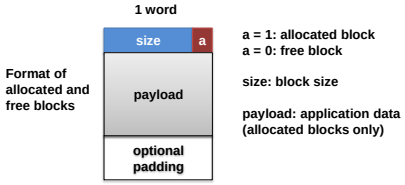


- **Explicit list** tracks only free blocks using pointers inside the free blocks

- **Segregated free list** – different free lists for different size classes
- **Blocks sorted by size**
  - Can use a balanced tree with pointers within each free block, and the length used as a key

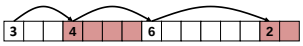
Method 1: Implicit List

- Need to identify whether each block is free or allocated
  - Can use extra bit
  - Bit can be put in the same word as the size if block sizes are always multiples of 2/4/8 (for alignment) – mask out low order bit when reading size

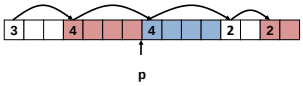


Implicit list: Allocating in free block

- Allocating from a free block – splitting
  - Since allocated space might be smaller than free space, we need to split the block

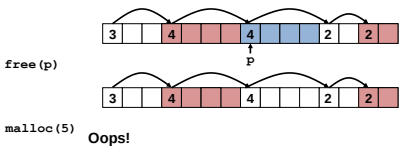


char * p = malloc(4);



Implicit list: Freeing a block

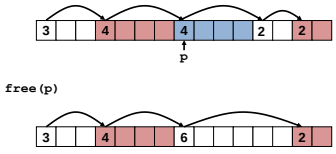
- Simplest implementation:
  - Only need to clear allocated flag
  - But can lead to "false fragmentation"



- There is enough free space, but the allocator won't be able to find it

Implicit list: Coalescing

- Join (coalesce) with next and/or previous block if free
  - Coalescing with next block

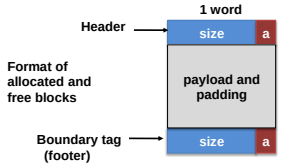


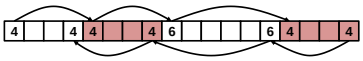
- But how do we coalesce with previous block?

Implicit list: Bidirectional coalescing

- **Boundary tags [Knuth73]**
  - Replicate size/allocated word at bottom of free blocks
  - Allows traversing a "list" backwards, but requires extra space
  - Important and general technique!

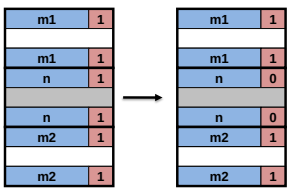
Donald Knuth
1938-





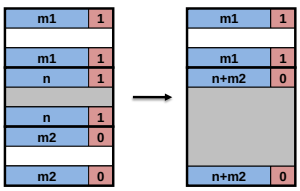
Constant time coalescing (Case 1)

- Both adjacent blocks are allocated
  - No coalescing is possible
  - Simple mark block free



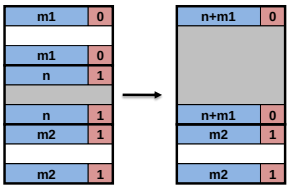
Constant time coalescing (Case 2)

- Merge current and next block
  - Update header of current and footer of next



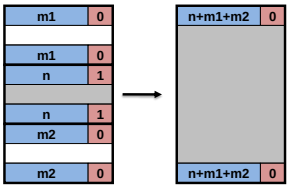
Constant time coalescing (Case 3)

- Previous block is merged with current
  - Update header of previous block and footer of current block



Constant time coalescing (Case 4)

- All three blocks are merged
  - Update header of previous and footer of next block



Overwriting memory

- Off-by-one errors
  - allocates N, operates on N+1

```
#define BUFSIZE 10

char * p = malloc(BUFSIZE);

// Add a NULL terminator to be safe
p[BUFSIZE] = 0;
```

What did we just overwrite?

Freeing blocks multiple times

```
x = malloc(N*sizeof(int));
<manipulate x>
free(x);

y = malloc(M*sizeof(int));
<manipulate y>
free(x);
```

Referencing freed blocks

```
x = malloc(N*sizeof(int));
<manipulate x>
free(x);
...
y = malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i]++;
```

Failing to free blocks (memory leaks)

- Slow, long-term killer

```
foo() {
   int *x = malloc(N*sizeof(int));
   ...
   return;
}
```