Today

- More memory allocation!

Fragmentation

- Poor memory utilization caused by fragmentation
  - internal fragmentation
  - external fragmentation

Internal Fragmentation

- For a given block, **internal fragmentation** occurs if payload is smaller than block size

- Caused by:
  - overhead of maintaining heap data structures (inside block, outside payload)
  - padding for alignment purposes
  - cyclic memory observers
    (e.g., to return a big block to satisfy a small request)
- Depends only on the pattern of previous requests
  - thus, easy to measure

External Fragmentation

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

External Fragmentation

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

Oops! (what would happen now?)
External Fragmentation
- Occurs when there is enough aggregate heap memory, but no single free block is large enough

\[ p_1 = \text{malloc}(4) \]
\[ p_2 = \text{malloc}(5) \]
\[ p_3 = \text{malloc}(6) \]
\[ \text{free}(p_2) \]
\[ p_4 = \text{malloc}(6) \]

*Oops! (what would happen now?)*

Implementation Issues
- How to know how much memory is being `free()`’d when it is given only a pointer (and no length)?
- How to keep track of the free blocks?
- What to do with extra space when allocating a block that is smaller than the free block it is placed in?
- How to pick a block to use for allocation—many might fit?
Knowing How Much to Free

- **Standard method**
  - Keep the 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.

Keeping Track of Free Blocks

- **Method 1:** *Implicit list* using length—links all blocks

- **Method 2:** *Explicit list* among the free blocks using pointers
Keeping Track of Free Blocks

- Method 1: **Implicit list** using length—links all blocks
  - ![Diagram showing linked blocks with length information]

- Method 2: **Explicit list** among the free blocks using pointers
  - ![Diagram showing explicit list]

- Method 3: **Segregated free list**
  - Different free lists for different size classes

- Method 4: **Blocks sorted by size**
  - Can use a balanced binary tree (e.g., red-black tree) with pointers within each free block, and the length used as a key

Implicit List

- For each block we need: length, is-allocated?
  - Could store this information in two words: wasteful!

- Standard trick
  - If blocks are aligned, some low-order address bits are always 0
  - Instead of storing an always-0 bit, use it as an allocated/free flag
  - When reading size word, must mask out this bit

![Format of allocated and free blocks]

Example

Sequence of blocks in heap: 2/0, 4/0, 18/0, 3/0

![Diagram of heap with blocks and allocation]

- 8-byte alignment
  - May require initial unused word
  - Causes some internal fragmentation

- One word (0/1) to mark end of list

- Here: block size in words for simplicity

Implicit List: Finding a Free Block

- First fit:
  - Search list from beginning, choose first free block that fits (Cost? Good?)

![Code snippet for first fit method]
Implicit List: Finding a Free Block

- First fit:
  - Search list from beginning, choose first free block that fits. (O(n^2))
  ```
  p = start;
  while (p < end or ( ( (p + 1) != end) && ( (p + 1) != 0 ) ) )
   \( \text{not passed end} \)
   p = p + 1;
  \( \text{already allocated} \)
  p = p + 1;
  \( \text{too small} \)
  p = p + 1;
  \( \text{go to next block (word addressed)} \)
  ```
  - Can take linear time in total number of blocks (allocated and free)
- Best fit:
  - Search the list, choose the best free block, fits, with fewest bytes left over
  - Keeps fragments small—usually helps fragmentation
  - Will typically run slower than first fit
- Next fit:
  - Like first fit, but search list starting where previous search finished
  - Should often be faster than first fit, avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse

Implicit List: Allocating in Free Block

- Allocating in a free block: splitting
  - Since allocated space might be smaller than free space, we might want to split the block. After allocating block of size 4.
  ```
  p
  4  4  4  4  4  4  4  4
  ```

- Allocating in a free block: splitting
  - Since allocated space might be smaller than free space, we might want to split the block. After allocating block of size 4.
  ```
  p
  4  4  4  4  4  4  4  4
  ```

void addBlock(p, 4) {
  int mask = (4 - 1); // round up to even
  int oldsize = (p + 2); // mark out low bit
  if (newsize < oldsize)
    *newsize = oldsize - newsize; // set length in remaining
  // part of block
}

Implicit List: Freeing a Block

- Simplest implementation?

```c
void freeBlock(int p) { *p = *p - 2; }`
Implicit List: Freeing a Block

- Simplest implementation:
  - Need only clear the "allocated" flag:
    ```
    void free(block *p) { p->allocated = 0; }
    ```
  - But can lead to "false fragmentation"

- alloc()! Oops!

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

How do we fix this?

Implicit List: Coalescing

- Join (coalesce) with next/previous blocks, if they are free
  - Coalescing with next block:
    ```
    void free(block *p) { p->allocated = 0; }
    if (p->next && p->next->allocated) { // add to this block if
      p->next = p; // find next block
    } else { // clear allocated flag
      p = p->next; // logically gone
    }
    ```

- Ehmm... But how do we coalesce with previous block?

 Implicit List: Bidirectional Coalescing

- Boundary tags [both?]
  - Replicate size/allocated used at "bottom" (end) of free blocks
  - Allows us to traverse the "list" backwards, but requires extra space
  - Important and general technique

Format of allocated and free blocks

Constant Time Coalescing

- Case 1: allocated
- Case 2: free
- Case 3: allocated
- Case 4: free

- Implementation: very simple
- Allocate cost:
  - Linear time worst case
- Free cost:
  - Constant time worst case
  - even with coalescing
- Memory usage:
  - will depend on placement policy
  - First-fit, next-fit or best-fit

- Not used in practice for malloc() / free() because of linear-time allocation
- used in many special purpose applications

- The concepts of splitting and boundary tag coalescing are general to all allocators
Keeping Track of Free Blocks

- **Method 1:** Implicit free list using length—links all blocks

- **Method 2:** Explicit free list among the free blocks using pointers

- **Method 3:** Segregated free list
  - Different free lists for different size classes

- **Method 4:** Blocks sorted by size
  - Can use a balanced tree (e.g., Red Black tree) with pointers within each free block, and the length used as a key

Explicit Free Lists

- **Logically (doubly-linked lists):**
  - ![Logical Diagram]

- **Physically: blocks can be in any order**
  - ![Physical Diagram]

Allocating From Explicit Free Lists

- **Before**
  - ![Before Diagram]

- **After**
  - ![After Diagram]
Freeing With Explicit Free Lists

- **Insertion policy**: Where in the free list do you put a newly freed block?
  - LIFO (last-in-first-out) policy
    - Insert freed block at the beginning of the free list
    - Pros?
    - Cons?
  - Address-ordered policy
    - Insert freed blocks so that free list blocks are always in address order:
      \[ \text{addr}(\text{pre}) < \text{addr}(\text{cur}) < \text{addr}(\text{next}) \]
    - Cons?
    - Pros?

Freeing With a LIFO Policy (Case 1)

Before

Root

\[ \text{free}(\bullet) \]

After

Freeing With a LIFO Policy (Case 2)

Before

Root

\[ \text{free}(\bullet) \]

After

Insert the freed block at the root of the list

Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list
Freeing With a LIFO Policy (Case 3)

Before
Root

After
Root

- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list

Explicit List Summary

- Comparison to implicit list:
  - Allocate is linear time in number of free blocks instead of all blocks
  - Much faster when most of the memory is full
  - Slightly more complicated allocate and free since needs to splice blocks in and out of the list
  - Some extra space for the links (2 extra words needed for each block)
  - Does this increase internal fragmentation?

- Most common use of linked lists is in conjunction with segregated free lists
  - Keep multiple linked lists of different size classes, or possibly for different types of objects

Freeing With a LIFO Policy (Case 4)

Before
Root

After
Root

- Splice out predecessor and successor blocks, coalesce all 3 memory blocks and insert the new block at the root of the list

Keeping Track of Free Blocks

- Method 1: Implicit list using length—links all blocks

- Method 2: Explicit list among the free blocks using pointers

- Method 3: Segregated free list
  - Different free lists for different size classes

- Method 4: Blocks sorted by size
  - Can use a balanced tree (e.g., Red-Black tree) with pointers within each free block, and the length used as a key

Segregated List (Seglist) Allocators

- Each size class of blocks has its own free list

- Often have separate classes for each small size
- For larger sizes: One class for each two-power size

Seglist Allocator

- Given an array of free lists, each one for some size class

- To allocate a block of size n:
  - Search appropriate free list for block of size m > n
  - If an appropriate block is found:
    - Split block and place fragment on appropriate list (optional)
  - If no block is found, try next larger class
  - Repeat until block is found

- If no block is found:
  - Request additional heap memory from OS using `malloc()`
  - Allocate block of n bytes from this new memory
  - Place remainder as a single free block in largest size class
Seglist Allocator (cont.)

- To free a block:
  - Coalesce and place on appropriate list (optional)

- Advantages of seglist allocators
  - Higher throughput
  - Log time for power-of-two size classes
  - Better memory utilization
  - First-fit search of segregated free list approximates a best-fit search of entire heap.
  - Extreme case: Giving each block its own size class is equivalent to best-fit.

Summary of Key Allocator Policies

- Placement policy:
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
  - Interesting observation: segregated free lists approximate a best-fit placement policy without having to search entire free list

- Splitting policy:
  - When do we go ahead and split free blocks?
  - How much internal fragmentation are we willing to tolerate?

- Coalescing policy:
  - Immediate coalescing: coalesce each time \texttt{free}() is called
  - Deferred coalescing: try to improve performance of \texttt{free}() by deferring coalescing until needed. Examples:
    - Coalesce as you scan the free list for \texttt{malloc}()
    - Coalesce when the amount of external fragmentation reaches some threshold

Implicit Memory Management: Garbage Collection

- Garbage collection: automatic reclamation of heap-allocated storage—application never has to free

```c
void free() {
  int *p = malloc(100);
  printf("p block is now garbage \n");
}
```

- Common in functional languages, scripting languages, and modern object oriented languages:
  - Lisp, ML, Java, Perl, Mathematica

- Variants (“conservative” garbage collectors) exist for C and C++
  - However, cannot necessarily collect all garbage

Garbage Collection

- How does the memory manager know when memory can be freed?
  - In general, we cannot know what is going to be used in the future since it depends on conditionals
  - But, we can tell that certain blocks cannot be used if there are no pointers to them

- Must make certain assumptions about pointers:
  - Memory manager can distinguish pointers from non pointers
  - All pointers point to the start of a block in the heap
  - Cannot hide pointers
    - e.g., by casting (coercing) them to an \texttt{int}, and then back again

Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
  - Does not move blocks (unless you also “compact”)

- Reference counting (Collins, 1960)
  - Does not move blocks (not discussed)

- Copying collection (Minsky, 1963)
  - Moves blocks (not discussed)

- Generational Collectors (Lieberman and Hewitt, 1983)
  - Collection based on lifetimes
  - Most allocations become garbage very soon
  - So focus reclamation work on zones of memory recently allocated

- For more information:

Memory as a Graph

- We view memory as a directed graph
  - Each block is a node in the graph
  - Each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called \texttt{root} nodes (e.g. registers, locations on the stack, global variables)

- A node (block) is \texttt{reachable} if there is a path from any root to that node
- Non-reachable nodes are \texttt{garbage} (cannot be needed by the application)
Mark and Sweep Collecting

- Can build on top of malloc/free package
  - Allocate using malloc until you “run out of space”
- When out of space:
  - Use extra mark bit in the head of each block
  - Mark: Start at roots and set mark bit on each reachable block
  - Sweep: Scan all blocks and free blocks that are not marked

### Before mark

### After mark

### After sweep

Assumptions For a Simple Implementation

- Application
  - new (n): returns pointer to new block with all locations cleared
  - read (b, i): read location i of block b into register
  - write (b, i, v): write v into location i of block b

- Each block will have a header word
  - Addressed as b[-1], for a block b

- Instructions used by the Garbage Collector
  - is_ptr (p): determines whether p is a pointer
  - length (b): returns the length of block b, not including the header
  - get_roots (): returns all the roots

Mark and Sweep (cont.)

Mark using depth-first traversal of the memory graph

```c
ptx_mark(p, ptr end) {
  if (!is_ptr(p)) return; // do nothing if not pointer
  if (markBit[p] == 0) return; // check if already marked
  markBit[p] = 1; // mark bit
  for (i=0; i < length[p]; i++) {
    // recursively call mark on all words in the block
    mark(p[i]);
    return;
  }
}
```

Sweep using lengths to find next block

```c
ptx_sweep(p, pte end) {
  while (p < pte end) {
    if (markBit[p] == 0) return;
    clearMarkBit ();
    if (allocated[p])
      free(p);
    p += length[p];
  }
}
```

Conservative Mark & Sweep in C

- A "conservative garbage collector" for C programs
  - is_ptr () determines if a word is a pointer by checking if it points to an allocated block of memory
  - But, in C pointers can point to the middle of a block

So how to find the beginning of the block?

- Can use a balanced binary tree to keep track of all allocated blocks
  - (Key is start of block)
  - (Balanced tree pointers can be stored in header (use two additional words))

Memory-Related Perils and Pitfalls

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks

Dereferencing Bad Pointers

The classic scanf bug

```c
int val;
...
scanf("%d", &val);
```
Reading Uninitialized Memory

- Assuming that heap data is initialized to zero

```c
/* return y = Ax */
int *matvec(int **A, int *x) {
    int *y = malloc( N * sizeof(int) );
    int i; 
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            y[i] = A[i][j] * x[j];
    return y;
}
```

Overwriting Memory

- Allocating the (possibly) wrong sized object

```c
int **p;
p = malloc( N * sizeof(int) );
for (i=0; i<N; i++) { 
p[i] = malloc( N * sizeof(int) );
}
```

Overwriting Memory

- Off-by-one error

```c
int **p;
p = malloc( N * sizeof(int *) );
for (i=0; i<N; i++) { 
p[i] = malloc( N * sizeof(int) );
}
```

Overwriting Memory

- Not checking the max string size

```c
char s[8];
int i;
gets(s); /* reads "123456789" from stdin */
```

- Basis for classic buffer overflow attacks
  - Your first assignment

Overwriting Memory

- Misunderstanding pointer arithmetic

```c
int *search(int *p, int val) {
    while (p && p != val)
        p += sizeof(int);
    return p;
}
```

Referencing Nonexistent Variables

- Forgetting that local variables disappear when a function returns

```c
int *foo () {
    int val;
    return &val;
}
```
Freeing Blocks Multiple Times

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

- What does the free list look like?
  ```c
  x = malloc( M * sizeof(int) );
  <manipulate x>
  free(x);
  free(x);
  ```

Freeing Blocks (Memory Leaks)

- Slow, silent, long-term killer!
  ```c
  foo() {
    int *a = malloc(M*sizeof(int));
    ...
    return;
  }
  ```

Referencing Freed Blocks

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

Failing to Free Blocks (Memory Leaks)

- Too much is reachable
  ```c
  Mark procedure is recursive
  * Will we have enough stack space?
  * We are garbage collecting because we are running out of memory, right?
  ```

Memory bugs?

Freeing only part of a data structure

```c
struct list {
  int val;
  struct list *next;
};

foo() {
  struct list *head = malloc( sizeof(struct list) );
  head->val = 0;
  head->next = NULL;
  <create and manipulate the rest of the list>
  ...
  free(head);
  return;
}
Overwriting Memory

- Referencing a pointer instead of the object it points to

```c
int *getPacket(int **packets, int *size) {
    int *packet;
    packet = packets[0];
    packets[0] = packets['size - 1'];
    *size--; // what is happening here?
    return (packet);
}
```

Dealing With Memory Bugs

- Conventional debugger (gdb)
  - Good for finding bad-pointer dereferences
  - Hard to detect the other memory bugs

- Debugging malloc (UToronto CSRI malloc)
  - Wrapper around conventional malloc
  - Detects memory bugs at malloc and free boundaries
  - Memory overwrites that corrupt heap structures
  - Some instances of freeing blocks multiple times
  - Memory leaks
  - Cannot detect all memory bugs
  - Overwrites into the middle of allocated blocks
  - Freeing block twice that has been reallocated in the interim
  - Referencing freed blocks