STACKS
Array and pointer

Allocate x bytes

\[
\text{sp} += x; \\
\text{return } \text{sp} - x;
\]
Stack Allocation

Array and pointer

Allocate x bytes

```
sp += x;
return sp - x;
```

Should check for stack overflow.
Stack Allocation

Array and pointer

Allocate x bytes

```
sp += x;
return sp - x;
```

Should check for stack overflow.
Array and pointer

Allocate x bytes

sp += x;
return sp - x;

Free x bytes

sp -= x;
Array and pointer

Allocate x bytes

Free x bytes

Should check for stack underflow.

```
sp += x;
return sp - x;
```

```
sp -= x;
```
Array and pointer

Allocate \( x \) bytes

\[
\text{sp} += x; \\
\text{return } \text{sp} - x;
\]

Free \( x \) bytes

\[
\text{sp} -= x;
\]

- Allocating and freeing take \( \Theta(1) \) time.
- Must free consistent with stack discipline.
- Limited applicability, but great when it works!
- One can allocate on the call stack using `alloca()`, but this function is deprecated, and the compiler is more efficient with fixed-size frames.
Stacks and Heaps

Stack

Heap

Image is in the public domain.
Fixed-size heap allocation
Heap Allocation*

C provides `malloc()` and `free()`. C++ provides `new` and `delete`.

Unlike Java and Python, C and C++ provide no garbage collector. Heap storage allocated by the programmer must be freed explicitly. Failure to do so creates a memory leak. Also, watch for dangling pointers and double freeing.

Memory checkers (e.g., AddressSanitizer, Valgrind) can assist in finding these pernicious bugs.

*Do not confuse with a heap data structure.
Free list

- Every piece of storage has the same size
- Unused storage has a pointer to next unused block

Bitmap mechanism
- Bit for each block saying whether or not it is free
- Bit tricks for allocation
Fixed–Size Allocation

Free list

Allocate 1 object

```c
x = free;
free = free->next;
return x;
```
Fixed–Size Allocation

Free list

Allocate 1 object

```c
x = free;
free = free->next;
return x;
```
Fixed-Size Allocation

Free list

Allocate 1 object

```c
x = free;
free = free->next;
return x;
```

Should check free != NULL.
Fixed-Size Allocation

Free list

Allocate 1 object

```c
x = free;
free = free->next;
return x;
```
Fixed-Size Allocation

Free list

Allocate 1 object

```c
x = free;
free = free->next;
return x;
```
Fixed-Size Deallocation

Free list

Allocate 1 object

```
x = free;
free = free->next;
return x;
```

free object x

```
x->next = free;
free = x;
```
Fixed–SizeDeallocation

Free list

Allocate 1 object

```c
x = free;
free = free->next;
return x;
```

free object x

```c
x->next = free;
free = x;
```
Fixed-Size Deallocation

Free list

Allocate 1 object

```c
x = free;
free = free->next;
return x;
```

free object x

```c
x->next = free;
free = x;
```
**Fixed-Size Deallocation**

**Free list**

Allocate 1 object

```
x = free;
free = free->next;
return x;
```

Free object \( x \)

```
x->next = free;
free = x;
```
Free Lists

Free list

Allocating and freeing take $\Theta(1)$ time.

Good temporal locality.

Poor spatial locality due to external fragmentation — blocks distributed across virtual memory — which can increase the size of the page table and cause disk thrashing.

The translation lookaside buffer (TLB) can also be a problem.
Mitigating External Fragmentation

- Keep a free list (or bitmap) per disk page.
- Allocate from the free list for the fullest page.
- Free a block of storage to the free list for the page on which the block resides.
- If a page becomes empty (only free-list items), the virtual-memory system can page it out without affecting program performance.

90–10 is better than 50–50:

\[
\text{Probability that 2 random accesses hit the same page} = 0.9 \times 0.9 + 0.1 \times 0.1 = 0.82 \text{ versus } 0.5 \times 0.5 + 0.5 \times 0.5 = 0.5
\]
VARIABLE-SIZE
HEAP ALLOCATION
Variable–Size Allocation

Binned free lists

- Leverage the efficiency of free lists.
- Accept a bounded amount of internal fragmentation.

Bin \( k \) holds memory blocks of size \( 2^k \).
Allocate for Binned Free Lists

Allocate x bytes

- If bin $k = \lfloor \lg x \rfloor$ is nonempty, return a block.
- Otherwise, find a block in the next larger nonempty bin $k' > k$, split it up into blocks of sizes $2^{k'-1}, 2^{k'-2}, \ldots, 2^k, 2^k$, and distribute the pieces.

Example

$x = 3 \Rightarrow \lfloor \lg x \rfloor = 2$. Bin 2 is empty.
Allocation for Binned Free Lists

Allocate \( x \) bytes

- If bin \( k = \lfloor \lg x \rfloor \) is nonempty, return a block.
- Otherwise, find a block in the next larger nonempty bin \( k' > k \), split it up into blocks of sizes \( 2^{k'-1}, 2^{k'-2}, \ldots, 2^k, 2^k \), and distribute the pieces.

Example

\( x = 3 \Rightarrow \lfloor \lg x \rfloor = 2 \).
Bin 2 is empty.
Allocate for Binned Free Lists

Allocate x bytes

- If bin $k = \lfloor \lg x \rfloor$ is nonempty, return a block.

- Otherwise, find a block in the next larger nonempty bin $k' > k$, split it up into blocks of sizes $2^{k'-1}, 2^{k'-2}, \ldots, 2^k, 2^k$, and distribute the pieces.

Example

$x = 3 \Rightarrow \lfloor \lg x \rfloor = 2$.
Bin 2 is empty.

return

© 2008–2018 by the MIT 6.172 Lecturers
### Allocation for Binned Free Lists

Allocate $x$ bytes

- If bin $k = \lfloor \lg x \rfloor$ is nonempty, return a block.
- Otherwise, find a block in the next larger nonempty bin $k' > k$, split it up into blocks of sizes $2^{k'-1}$, $2^{k'-2}$, ..., $2^k$, $2^k$, and distribute the pieces.*

#### Example

$x = 3 \Rightarrow \lfloor \lg x \rfloor = 2$.

Bin 2 is empty.

*If no larger blocks exist, ask the OS to allocate more memory.*
Storage Layout of a Program

- **high address**
- **virtual memory**
- **low address**

- **stack**
- **heap**
- **bss**
- **data**
- **text**

- Dynamically allocated
- Initialized to 0 at program start
- Read from disk
- Code

© 2008–2018 by the MIT 6.172 Lecturers
Q. Since a 64-bit address space takes over a century to write at a rate of 4 billion bytes per second, we effectively never run out of virtual memory. Why not just allocate out of virtual memory and never free?

A. **External fragmentation** would be horrendous! The performance of the page table would degrade tremendously leading to disk thrashing, since all nonzero memory must be backed up on disk in page-sized blocks.

**Goal of storage allocators**
Use as little virtual memory as possible, and try to keep the used portions relatively compact.
Theorem. Suppose that the maximum amount of heap memory in use at any time by a program is $M$. If the heap is managed by a BFL allocator, the amount of virtual memory consumed by heap storage is $O(M \lg M)$.

Proof. An allocation request for a block of size $x$ consumes $2^{\lceil \lg x \rceil} \leq 2x$ storage. Thus, the amount of virtual memory devoted to blocks of size $2^k$ is at most $2M$. Since there are at most $\lg M$ free lists, the theorem holds.

⇒ In fact, BFL is $\Theta(1)$–competitive with the optimal allocator (assuming no coalescing).
Coalescing

Binned free lists can sometimes be heuristically improved by splicing together adjacent small blocks into a larger block.

- Clever schemes exist for finding adjacent blocks efficiently — e.g., the “buddy” system — but the overhead is still greater than simple BFL.
- No good theoretical bounds exist that prove the effectiveness of coalescing.
- Coalescing seems to reduce fragmentation in practice, because heap storage tends to be deallocated as a stack (LIFO) or in batches.
GARBAGE COLLECTION BY REFERENCE COUNTING
Garbage Collectors

Idea

- Free the programmer from freeing objects.
- A garbage collector identifies and recycles the objects that the program can no longer access.
- GC can be built-in (Java, Python) or do-it-yourself.
**Terminology**

- **Roots** are objects directly accessible by the program (globals, stack, etc.).
- **Live** objects are reachable from the roots by following pointers.
- **Dead** objects are inaccessible and can be recycled.

**How can the GC identify pointers?**

- Strong typing.
- Prohibit pointer arithmetic (which may slow down some programs).
Reference Counting

Keep a count of the number of pointers referencing each object. If the count drops to 0, free the dead object.
Reference Counting

Keep a count of the number of pointers referencing each object. If the count drops to 0, free the dead object.
Reference Counting

Keep a count of the number of pointers referencing each object. If the count drops to 0, free the dead object.
Reference Counting

Keep a count of the number of pointers referencing each object. If the count drops to 0, free the dead object.
Reference Counting

Keep a count of the number of pointers referencing each object. If the count drops to 0, free the dead object.
Reference Counting

Keep a count of the number of pointers referencing each object. If the count drops to 0, free the dead object.
Limitation of Reference Counting

Problem
A cycle is never garbage collected!

![Diagram showing a cycle in a reference graph](image)
Limitation of Reference Counting

Problem
A cycle is never garbage collected!
Problem
A cycle is never garbage collected!

Nevertheless, reference counting works well for acyclic structures.
MARK–AND–Sweep
GARBAGE COLLECTION
**Graph Abstraction**

**Idea**
Objects and pointers form a directed graph $G = (V, E)$. Live objects are reachable from the roots. Use breadth-first search to find the live objects.

```plaintext
for (∀ v ∈ V) {
    if (root(v)) {
        v.mark = 1;
        enqueue(Q, v);
    } else v.mark = 0;
}
while (Q != ∅) {
    u = dequeue(Q);
    for (∀ v ∈ V such that (u,v) ∈ E) {
        if (v.mark == 0) {
            v.mark = 1;
            enqueue(Q, v);
        }
    }
}
```

**FIFO queue Q**

![FIFO queue diagram]
Breadth-First Search

Q

head
tail

© 2008–2018 by the MIT 6.172 Lecturers
Breadth–First Search

Q

head
tail
Breadth-First Search

The diagram illustrates the Breadth-First Search algorithm on a graph. The algorithm starts at the root node (r) and explores all the neighbor nodes at the present depth before moving on to the nodes at the next depth level. The queue Q contains the nodes in the order they are visited, with the head of the queue being the next node to be explored.
Breadth-First Search

A breadth-first search (BFS) is a search algorithm that explores vertices of a graph in breadth-first order. It starts at the root node and explores all the neighboring nodes before moving to the next level of nodes.

In the diagram, the nodes are represented as circles, and the edges represent the connections between them. The queue (Q) is used to keep track of the nodes to be visited, with the head and tail pointers indicating the current position of the search.

The BFS algorithm visits the nodes in the following order:
1. The root node (r).
2. The nodes directly connected to the root (b).
3. The nodes directly connected to b (c, d, e).
4. The nodes directly connected to c, d, and e (f, g, h).
5. The nodes directly connected to f, g, and h (i, j).

This process continues until all nodes have been visited.
Breadth-First Search
Breadth-First Search

Q

head  tail

r  b  c

© 2008–2018 by the MIT 6.172 Lecturers
Breadth-First Search

© 2008–2018 by the MIT 6.172 Lecturers
Breadth-First Search
Breadth–First Search
Breadth-First Search

Q: r b c d e

head tail
Breadth-First Search

Q: r b c d e f

head tail

© 2008–2018 by the MIT 6.172 Lecturers
Breadth-First Search
Breadth-First Search
Breadth-First Search

![Breadth-First Search Diagram]

Q: [r b c d e f g]

head  tail

© 2008–2018 by the MIT 6.172 Lecturers
Breadth-First Search

![Diagram of a graph with nodes labeled a, b, c, d, e, f, g, h, i, j, and a queue Q containing nodes r, b, c, d, e, f, g. The diagram shows the order in which nodes are visited, with arrows indicating the search order. The queue is highlighted with an arrow pointing to the head and tail.]
**Mark stage:** Breadth-first search marked all of the live objects.

**Sweep stage:** Scan over memory to free unmarked objects.

Mark–and–sweep doesn’t deal with fragmentation
STOP–AND–COPY
GARBAGE COLLECTION
Observation
All live vertices are placed in contiguous storage in Q.
Copying Garbage Collector

FROM space

next allocation

live  
dead  
unused
Copying Garbage Collector

FROM space

next allocation

live
dead
unused
Copying Garbage Collector

FROM space

next allocation
Copying Garbage Collector

FROM space

next allocation

live
dead
unused
Copying Garbage Collector

FROM space

next allocation

live
dead
unused
Copying Garbage Collector

FROM space

next allocation

live
dead
unused
When the **FROM** space is “full,” copy live storage using BFS with the **TO** space as the FIFO queue.
When the **FROM** space is “full,” copy live storage using BFS with the **TO** space as the FIFO queue.

© 2008–2018 by the MIT 6.172 Lecturers
Updating Pointers

Since the **FROM** address of an object is not generally equal to the **TO** address of the object, pointers must be updated.

- When an object is copied to the **TO** space, store a forwarding pointer in the **FROM** object, which implicitly marks it as moved.
- When an object is removed from the FIFO queue in the **TO** space, update all its pointers.
Remove an item from the queue.
Remove an item from the queue.
Example

Enqueue adjacent vertices.
Enqueue adjacent vertices.
Place forwarding pointers in FROM vertices.
Update the pointers in the removed item to refer to its adjacent items in the **TO** space.
Update the pointers in the removed item to refer to its adjacent items in the **TO** space.
Example

Linear time to copy and update all vertices.
When Is the FROM Space “Full”?

FROM

| used | heap |

- Request new heap space equal to the used space, and consider the FROM space to be “full” when this heap space has been allocated.
- The cost of garbage collection is then proportional to the size of the new heap space \( \Rightarrow \) amortized \( O(1) \) overhead, assuming that the user program touches all the memory allocated.
- Moreover, the VM space required is \( O(1) \) times optimal by locating the FROM and TO spaces in different regions of VM where they cannot interfere with each other.
Dynamic Storage Allocation

Lots more is known and unknown about dynamic storage allocation. Strategies include

- buddy system,
- variants of mark–and–sweep,
- generational garbage collection,
- real–time garbage collection,
- multithreaded storage allocation,
- parallel garbage collection,
- etc.
Summary

- Stack: most basic form of storage and is very efficient when it works
- Heap is the more general form of storage
- Fixed-size allocation using free lists
- Variable-sized allocation using binned free lists
- Garbage collection – reference counting, mark-and-sweep, stop-and-copy
- Internal and external fragmentation
- You will look at storage allocation in Homework 6 and Project 3