The Hardware/Software Interface
CSE351 Winter 2013

Memory Allocation I
**Roadmap**

**C:**
```c
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

**Java:**
```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();
```

**Assembly language:**
```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

**Machine code:**
```
0111010000011000
1000110100000100
1000100111000010
110000001111101000011111
```

**OS:**
- Windows 8
- Mac

**Data & addressing**
- Integers & floats

**Machine code & C**
- x86 assembly

**Programming**
- Procedures & stacks

**Arrays & structs**
- Memory & caches

**Processes**
- Virtual memory

**Java vs. C**
- Memory allocation
- Winter 2013
Memory Allocation Topics

- **Dynamic memory allocation**
  - Size/number of data structures may only be known at run time
  - Need to allocate space on the heap
  - Need to de-allocate (free) unused memory so it can be re-allocated

- **Implementation**
  - Implicit free lists
  - Explicit free lists – subject of next programming assignment
  - Segregated free lists

- **Garbage collection**

- **Common memory-related bugs in C programs**
Dynamic Memory Allocation

- Programmers use **dynamic memory allocators** (such as `malloc`) to acquire VM at run time.
  - For data structures whose size is only known at runtime.

- **Dynamic memory allocators** manage an area of process virtual memory known as the **heap**.
Dynamic Memory Allocation

- Allocator maintains heap as collection of variable sized blocks, which are either allocated or free
  - Allocator requests space in heap region; VM hardware and kernel allocate these pages to the process
  - Application objects are typically smaller than pages, so the allocator manages blocks within pages

- Types of allocators
  - **Explicit allocator**: application allocates and frees space
    - E.g. `malloc` and `free` in C
  - **Implicit allocator**: application allocates, but does not free space
    - E.g. garbage collection in Java, ML, and Lisp
The `malloc` Package

```c
#include <stdlib.h>

void *malloc(size_t size)
```

- **Successful:**
  - Returns a pointer to a memory block of at least `size` bytes (typically) aligned to 8-byte boundary
  - If `size == 0`, returns NULL
- **Unsuccessful:** returns NULL and sets `errno`

```c
void free(void *p)
```

- Returns the block pointed at by `p` to pool of available memory
- `p` must come from a previous call to `malloc` or `realloc`

**Other functions**

- `calloc`: Version of `malloc` that initializes allocated block to zero.
- `realloc`: Changes the size of a previously allocated block.
- `sbrk`: Used internally by allocators to grow or shrink the heap.
void foo(int n, int m) {
    int i, *p;

    /* allocate a block of n ints */
    p = (int *)malloc(n * sizeof(int));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }
    for (i=0; i<n; i++) p[i] = i;

    /* add space for m ints to end of p block */
    if (((p = (int *)realloc(p, (n+m) * sizeof(int))) == NULL) {
        perror("realloc");
        exit(0);
    }
    for (i=n; i < n+m; i++) p[i] = i;

    /* print new array */
    for (i=0; i<n+m; i++)
        printf("%d\n", p[i]);

    free(p); /* return p to available memory pool */
}
Assumptions Made in This Lecture

- Memory is word addressed (each word can hold a pointer)
  - block size is a multiple of words

![Diagram of memory allocation with allocated and free blocks]
Allocation Example

\[ p1 = \text{malloc}(4) \]

\[ p2 = \text{malloc}(5) \]

\[ p3 = \text{malloc}(6) \]

\[ \text{free}(p2) \]

\[ p4 = \text{malloc}(2) \]
How are going to implement that?!?

- Ideas?
Constraints

**Applications**
- Can issue arbitrary sequence of malloc() and free() requests
- free() requests must be made only for a previously malloc()’d block

**Allocators**
- Can’t control number or size of allocated blocks
- Must respond immediately to malloc() requests
  - *i.e.*, can’t reorder or buffer requests
- Must allocate blocks from free memory
  - *i.e.*, blocks can’t overlap
- Must align blocks so they satisfy all alignment requirements
  - 8 byte alignment for GNU malloc (**libc** malloc) on Linux boxes
- Can’t move the allocated blocks once they are malloc()’d
  - *i.e.*, compaction is not allowed. *Why not?*
Performance Goal: Throughput

- Given some sequence of `malloc` and `free` requests:
  - \( R_0, R_1, ..., R_k, ..., R_{n-1} \)

- Goals: maximize throughput and peak memory utilization
  - These goals are often conflicting

- Throughput:
  - Number of completed requests per unit time
  - Example:
    - 5,000 `malloc()` calls and 5,000 `free()` calls in 10 seconds
    - Throughput is 1,000 operations/second
Performance Goal: Peak Memory Utilization

- Given some sequence of malloc and free requests:
  - \( R_0, R_1, ..., R_k, ..., R_{n-1} \)

- **Def**: Aggregate payload \( P_k \)
  - \( \text{malloc}(p) \) results in a block with a **payload** of \( p \) bytes
  - After request \( R_k \) has completed, the aggregate payload \( P_k \) is the sum of currently allocated payloads

- **Def**: Current heap size = \( H_k \)
  - Assume \( H_k \) is monotonically nondecreasing
    - Allocator can increase size of heap using \( \text{sbrk}() \)

- **Def**: Peak memory utilization after \( k \) requests
  - \( U_k = \left( \max_{i<k} P_i \right) \ / \ H_k \)
  - Goal: maximize utilization for a sequence of requests.
  - **Why is this hard? And what happens to throughput?**
Fragmentation

- Poor memory utilization is caused by *fragmentation*
  - *internal* fragmentation
  - *external* fragmentation
Internal Fragmentation

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

  ![Diagram](image)

  - **Caused by**
    - overhead of maintaining heap data structures (inside block, outside payload)
    - padding for alignment purposes
    - explicit policy decisions (e.g., to return a big block to satisfy a small request)
      *why would anyone do that?*

  - **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

```
p1 = malloc(4)  
p2 = malloc(5)  
p3 = malloc(6)  
free(p2)        
p4 = malloc(6)  
```

**Oops! (what would happen now?)**

- Depends on the pattern of future requests
  - Thus, difficult to measure