Memory Allocation I

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
}

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.
```

Assumptions Made in This Lecture

- Memory is word addressed (each word can hold a pointer)
  - block size is a multiple of words
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
    - \textit{i.e.}, can’t reorder or buffer requests
  - Must allocate blocks from free memory
    - \textit{i.e.}, blocks can’t overlap
  - Must align blocks so they satisfy all alignment requirements
    - \textit{i.e.}, blocks can’t overlap
  - 8 byte alignment for GNU malloc (\texttt{libc malloc}) on Linux boxes
  - Can’t move the allocated blocks once they are malloc()’d
    - \textit{i.e.}, compaction is not allowed. \textit{Why not?}

Performance Goal: Throughput

- Given some sequence of \texttt{malloc} and \texttt{free} requests:
  - \( R_0, R_1, ..., R_n \)

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

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

- Given some sequence of \texttt{malloc} and \texttt{free} requests:
  - \( R_0, R_1, \ldots, R_n \)

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

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

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

### Internal Fragmentation

- For a given block, \textit{internal fragmentation} occurs if payload is smaller than block size

- \textbf{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)

- \textbf{Depends only on the pattern of previous requests}
  - thus, easy to measure

### External Fragmentation

- \textbf{Omits} when there is enough aggregate heap memory, but no single free block is large enough

- \textbf{Depends on the pattern of future requests}
  - Thus, difficult to measure

- \texttt{malloc(4)}
- \texttt{malloc(5)}
- \texttt{malloc(6)}
- \texttt{free(p2)}
- \texttt{malloc(6)}

- \textit{Oops! (what would happen now?)}