Dynamic Memory Allocation: Basic Concepts

CSC 235 - Computer Organization

References

- Slides adapted from CMU

Dynamic Memory Allocation

- Programmers use dynamic memory allocators (such as `malloc`) to acquire virtual memory (VM) at run time.
  - For data structures where the size is only known at runtime
- Dynamic memory allocators manage an area of process VM known as the heap.

Dynamic Memory Allocation

- Allocator maintains the heap as a collection of variable sized blocks, which are either allocated or free.
- Types of allocators
  - Explicit allocator: application allocates and frees space (for example, `malloc` and `free` in C)
  - Implicit allocator: application allocates, but does not free space (for example, `new` and garbage collection in Java)
- This lecture: explicit memory allocation

The `malloc` Package

- `void *malloc(size_t size)`
  - Success: returns a pointer to a memory block of at least `size` bytes aligned to a 16-byte boundary (on x86-64); if `size == 0`, returns `NULL`
  - Unsuccessful: returns `NULL` and sets `errno` to `ENOMEM`
- `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`, `calloc`, 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

`malloc` Example

```
#include <stdio.h>
#include <stdlib.h>

void foo(long n) {
    long i, *p;

    /* Allocate a block of n longs */
    p = (long *) malloc(n * sizeof(long));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }

    /* Initialize allocated block */
    for (i=0; i<n; i++)
        p[i] = i;
    /* Do something with p */
```

Sample Implementation

- Code (location: CS:APP3e Code Examples
  - File: `mm.c`
  - Manges fixed size heap
  - Functions `mm_malloc` and `mm_free`
- Features
  - Based on words of 8 bytes each
  - Pointers returned by `malloc` are double-word aligned
  - Compile and run tests with command interpreter

Constraints

- Applications
  - Can issue arbitrary sequence of `malloc` and `free` requests
  - `free` request must be to a `malloc`'d block
- Explicit Allocators
  - Cannot control number or size of allocated blocks
  - Must respond immediately to `malloc` requests
  - Must allocate blocks from free memory
  - Must align blocks to satisfy alignment requirements
  - Can manipulate and modify only free memory
  - Cannot move the allocated blocks once they are `malloc`'d

Performance Goal: Throughput

- Given some sequence of `malloc` and `free` requests:

$$R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$$

- 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 per second

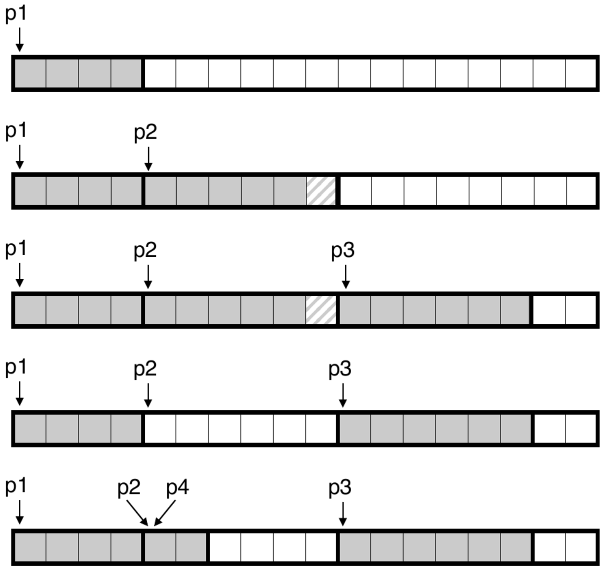
Performance Goal: Minimize Overhead

- Given some sequence of `malloc` and `free` requests:

$$R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$$

- Definition: aggregate payload P_k
  - `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
- Definition: current heap size H_k
  - Assume H_k is monotonically non-decreasing, that is, the heap only grows when the allocator uses `sbrk`
- Definition: Overhead after $k + 1$ requests
  - Fraction of heap space not used for program data
  - $O_k = H_k / (max_{i \le k} P_i) - 1$

`malloc` Heap Visualization Example



Fragmentation

- Fragmentation causes poor memory utilization
- Internal fragmentation: For a given block, internal fragmentation occurs if payload is smaller than block size
  - Caused by
    - overhead of maintaining heap data structures
    - padding for alignment purposes
    - explicit policy decisions (for example, to return a big block to satisfy a small request)
  - Depends only on the pattern of previous requests
- External fragmentation: occurs when there is enough aggregate heap memory, but no single free block is large enough
  - Amount of external fragmentation depends on the pattern of future requests (difficult to measure)

Implementation Issues

- How do we know how much memory to free given only 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 place?
- How do we pick a block to use for allocation – many might fit?
- How do we reuse a block that has been freed?

Knowing How Much to Free

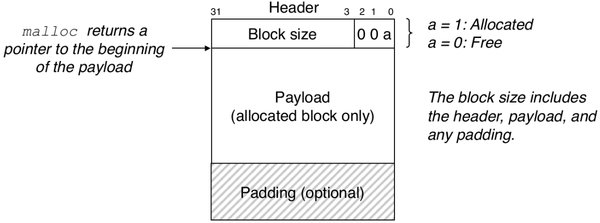
- Standard method
  - Keep the length (in bytes) of a block in the word preceding the block, including the header
  - Requires an extra word for every allocated block

Keeping Track of Free Blocks

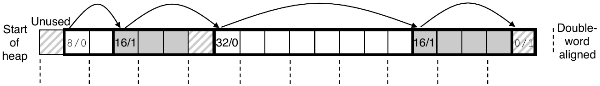
- Method 1: Implicit list using length; links all blocks
  - Need to tag each block as allocated/free
- Method 2: Explicit list among the free blocks using pointers
  - Need space for pointers
- Method 3: Segregated free list
  - Different free lists for different size classes
- Method 4: Blocks sorted by size
  - Can use a balanced tree with pointers within each free block, and the length used as a key

Method 1: Implicit Free List

- For each block we need both size and allocation status
  - Could store this information in two words (wasteful)
- Standard trick
  - When blocks are aligned, some low-order address bits are always zero
  - Instead of storing the always zero bit, use it as an allocated/free flag
  - When reading the size word, the bit must be masked out



Detailed Implicit Free List Example



- Allocated blocks: shaded
- Free blocks: unshaded
- Headers: labeled with "size in words/allocated bit"
  - Headers are at non-aligned positions
  - Payloads are aligned

Implicit List: Data Structures

- Block declaration

```
typedef unint64_t word_t;

typedef struct block {
    word_t header;
    unsigned char payload[0]; // zero length array
} block_t;
```

- Getting payload from block pointer

```
return (void *) (block->payload);
```

- Getting header from payload

```
return (void *) ((unsigned char *) bp - offsetof(block_
```

Implicit List: Header access

- Getting allocated bit from header

```
return header & 0x1;
```

- Getting size from header

```
return header & ~0xfL;
```

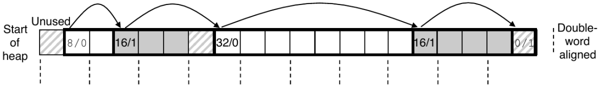
- Initializing header

```
block->header = size | alloc;
```

Implicit List: Traversing the List

- Find next block

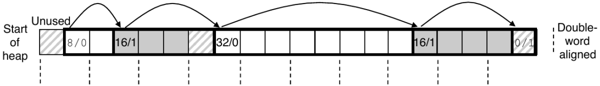
```
static block_t *find_next(block_t *block) {
    return (block_t *) ((unsigned char *) block
                        + get_size(block));
}
```



Implicit List: Finding a Free Block

- Search list from beginning and choose first free block that fits (including space for the header)

```
static block_t *find_fit(size_t asize) {
    block_t *block;
    for (block = heap_start; block != heap_end;
         block = find_next(block))
    {
        if (!(get_alloc(block)) && (asize <= get_size(block
            return block;
    }
    return NULL; // No fit found
}
```

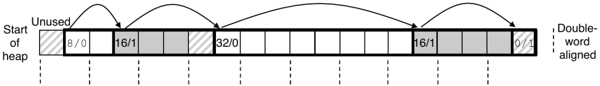


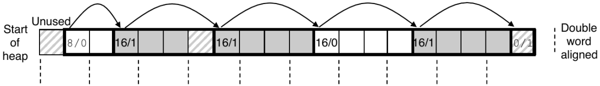
Implicit List: Finding a Free Block

- First fit:
  - Search list from the beginning and choose the first free block that fits
  - Can take linear time in total number of blocks (allocated and free)
  - In practice it can cause "splinters" at the beginning of the list
- Next fit:
  - Like first fit, but search the list starting where the previous search finished
  - Should often be faster than first fit since it avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse
- Best fit:
  - Search the list and choose the best free block: fits with the fewest bytes left over
  - Keeps fragments small; usually improves memory utilization
  - Will typically run slower than first fit
  - Still a greedy algorithm; no guarantee of optimality

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



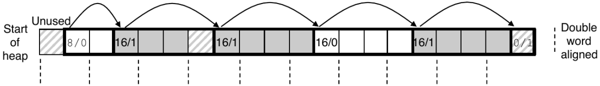


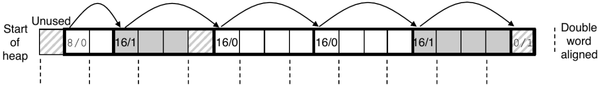
Implicit List: Splitting Free Block

```
// Warning: This code is incomplete

static void split_block(block_t *block, size_t asize) {
    size_t block_size = get_size(block);

    if ((block_size - asize) >= min_block_size) {
        write_header(block, asize, true);
        block_t *block_next = find_next(block);
        write_header(block_next, block_size - asize, false
    }
}
```

Implicit List: Freeing a Block

- Simplest implementation:
  - Need to clear the "allocated" flag
  - But, can lead to "false fragmentation"



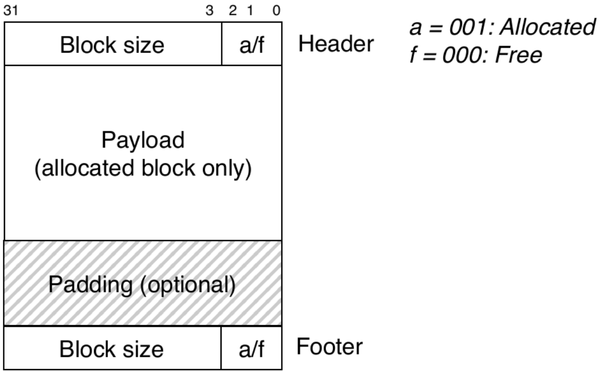


Implicit List: Coalescing

- Join (coalesce) with next/previous blocks, if they are free
- Coalesce with next block
  - Simple because of forward search
- How do we coalesce with previous block?
  - How do we know where it starts?
  - How can we determine whether it is allocated?

Implicit List: Bidirectional Coalescing

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



Implementation with Footers

- Locating footer of current block

```
const size_t dsize = 2 * sizeof(word_t);

static word_t *header_to_footer(block_t *block) {
    size_t asize = get_size(block);
    return (word_t *) (block->payload + asize - dsize);
}
```

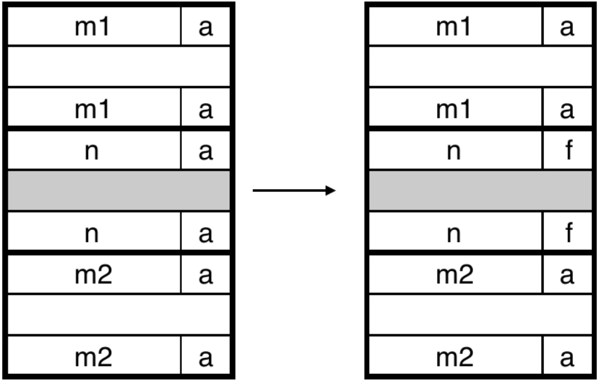
- Locating footer of previous block

```
static word_t *find_prev_footer(block_t *block) {
    return &(block->header) - 1);
}
```

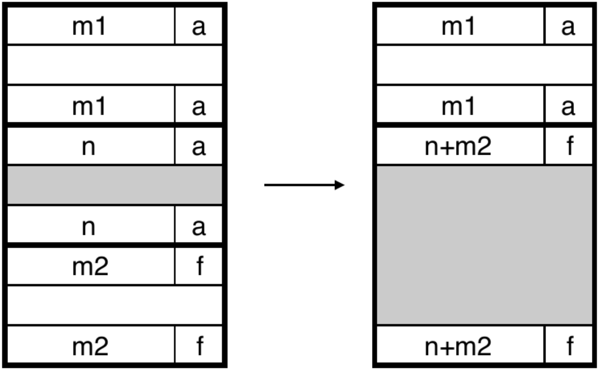
Splitting Free Block: Full Version

```
static void split_block(block_t *block, size_t asize) {
    size_t block_size = get_size(block);

    if ((block_size - asize) >= min_block_size) {
        write_header(block, asize, true);
        write_footer(block, asize, true);
        block_t *block_next = find_next(block);
        write_header(block_next, block_size - asize, false)
        write_footer(block_next, block_size - asize, false)
    }
}
```

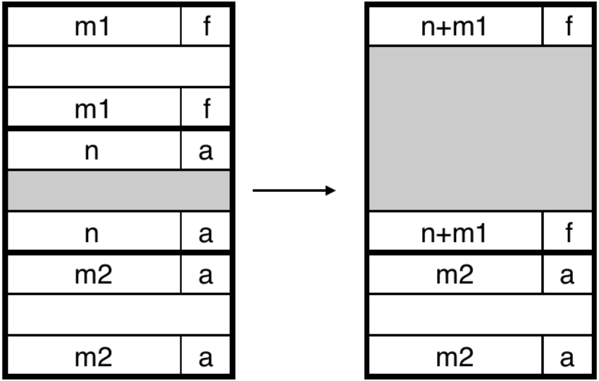
Constant Time Coalescing (Case 1)



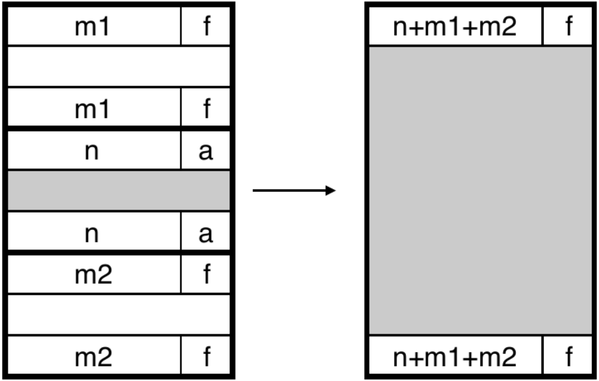
Constant Time Coalescing (Case 2)



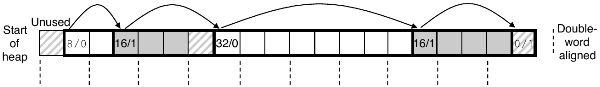
Constant Time Coalescing (Case 3)



Constant Time Coalescing (Case 4)



Heap Structure



- Dummy footer before first header
  - Marked as allocated
  - Prevents accidental coalescing when freeing first block
- Dummy header after last footer
  - Prevents accidental coalescing when freeing final block

Top-Level Malloc Code

```
const size_t dsize = 2*sizeof(word_t);

void *mm_malloc(size_t size)
{
    size_t asize = round_up(size + dsize, dsize);

    block_t *block = find_fit(asize);

    if (block == NULL)
        return NULL;

    size_t block_size = get_size(block);
    write_header(block, block_size, true);
    write_footer(block, block_size, true);

    split_block(block, asize);
```

Top-Level Free Code

```
void mm_free(void *bp)
{
    block_t *block = payload_to_header(bp);
    size_t size = get_size(block);

    write_header(block, size, false);
    write_footer(block, size, false);

    coalesce_block(block);
}
```

Disadvantages of Boundary Tags

- Internal fragmentation
- Can it be optimized?
  - Which blocks need the footer tag?
  - What does that mean?

No Boundary Tag for Allocated Blocks

- Boundary tag needed only for free blocks
- When sizes are multiples of 16, have 4 spare bits
- Header: Use 2 bits (address bits always zero due to alignment):

  `(prev_block) << 1 | (curr_block)`

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 (next lecture) 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 `free` is called
  - Deferred coalescing: try to improve performance of `free` by deferring coalescing until needed

Implicit Lists: Summary

- Implementation: very simple
- Allocate cost: linear time worst case
- Free cost: constant time worst case (even with coalescing)
- Memory overhead: depends on placement policy
- Not used in practice for `malloc`/`free` because of linear time allocation
- The concepts of splitting and boundary tag coalescing are general to all allocators