CSE320 System Fundamentals II Dynamic Memory Allocation

YOUNGMIN KWON / TONY MIONE

Announcements

Reading: Text 9.9-9.11

Exam 1 :

- Thur, 10/17/2024 – During class period
- Covers through this lecture!
- Closed book but A4 note sheet written in your own hand (both sides) is allowed during test.

Acknowledgements

Some slides provided by Dr Yoon Seok Yang

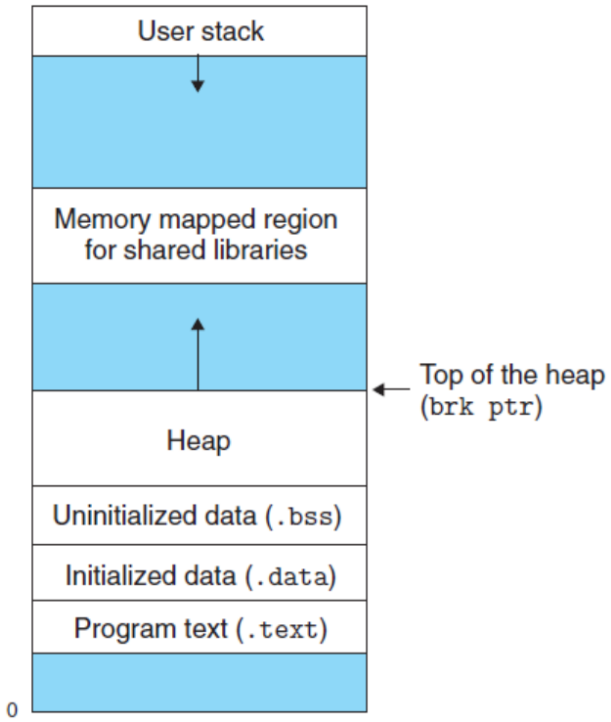
Dynamic Memory Allocation

A dynamic memory allocator maintains an area of a process's virtual memory known as the heap

For each process, the kernel maintains a variable brk that points to the top of the heap.

Explicit allocator
- malloc, free

Implicit allocator
- 'new'
- Garbage collectors



The brk() function in C is a system call that historically was used to set the end of the data segment (heap) of a process's address space.

Why Virtual Memory?

Virtual memory is a memory management technique that abstracts and manages physical memory in a way that provides several advantages

- **Abstraction of Physical Memory**: Virtual memory abstracts the actual physical memory (RAM) of the computer. It allows applications to operate as if they have access to a large and contiguous block of memory, even if the physical RAM is smaller or fragmented.
- **Address Spaces**: Each running process (application) has its own virtual address space, which is a range of memory addresses that it can use. These addresses are referred to as virtual addresses.
- **Memory Protection**: Virtual memory systems provide memory protection mechanisms. Each process is isolated from other processes, and the operating system ensures that one process cannot access the memory of another process, protecting both the stability and security of the system.
- **Efficient Memory Utilization**: Virtual memory enables efficient memory utilization by allowing the operating system to optimize the use of physical memory and disk space. Frequently used data can be kept in RAM, while less frequently used data can be moved to disk.
- **Address Space Isolation**: Virtual memory allows each process to have its own address space, which provides address space isolation. This means that one process cannot accidentally access the memory of another process.

malloc and free

void *malloc(size_t size) allocates at least the request size bytes

- calloc works like malloc but initializes the memory to 0 in addition

void free(void *ptr) frees the allocated memory

void *sbrk(intptr_t incr) increases brk by incr and returns the old brk.

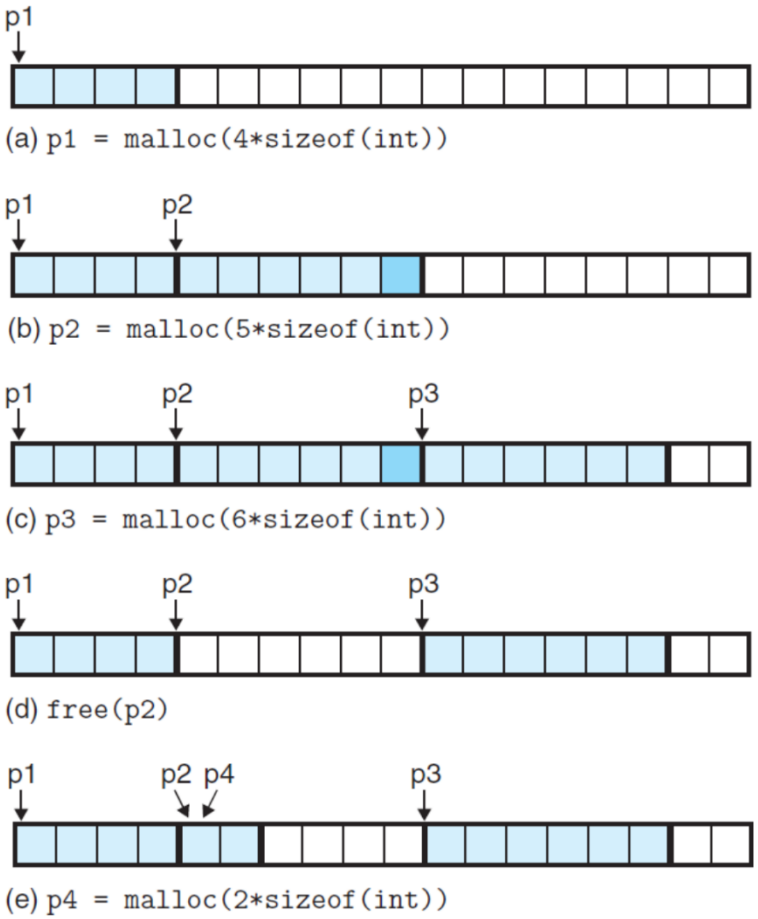
sbrk () vs malloc()

sbrk()

- The sbrk() function in C is a system call that allows a program to adjust the size of its data segment.
- It is not a standard C library function and is specific to Unix-like operating systems.
- Allocating memory
  - void *new_memory = sbrk(increment);
- Deallocating memory
  - sbrk(-decrement);
  - It can lead to fragmentation issues if not managed properly.
- Error Handling
  - sbrk() returns (void *)-1 in case of an error,

Modern C programs often use higher-level memory management functions like malloc(), free(), and calloc() from the C standard library or memory management libraries like the C++ Standard Library.

malloc and free



each cell: 2 byte
int: 2 byte
alignment: 4 byte

Dark blue area is a padding for the alignment

Fragmentation

Internal fragmentation

- Allocated blocks are larger than payloads
- Due to minimum size constraints on allocation
- Due to padding for the alignment.
  - E.g. the dark blue cell in (b), (c) of prev. page

External fragmentation

- There is enough aggregate free memory to satisfy the request, but no single block is large enough.
  - E.g. malloc(5*sizeof(int)) after (e) of prev. page

Implementation Issues

Free block organization
- How to keep track of free blocks?

Placement
- How to choose a free block for a request?

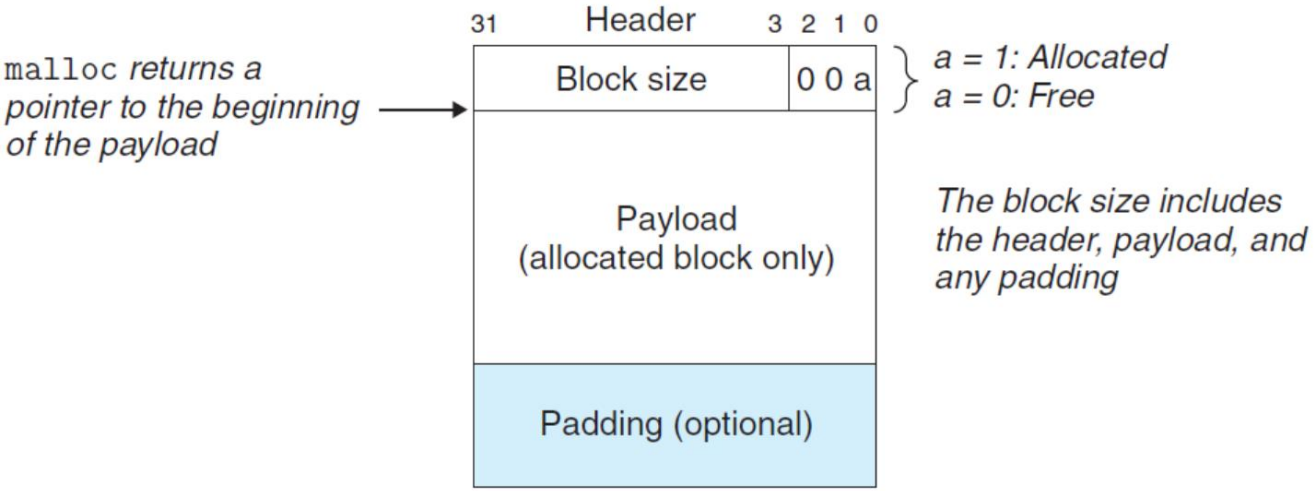
Splitting
- When a part of a free block is allocated, what do we do for the remaining free blocks?

Coalescing
- What do we do with a block that is just freed?

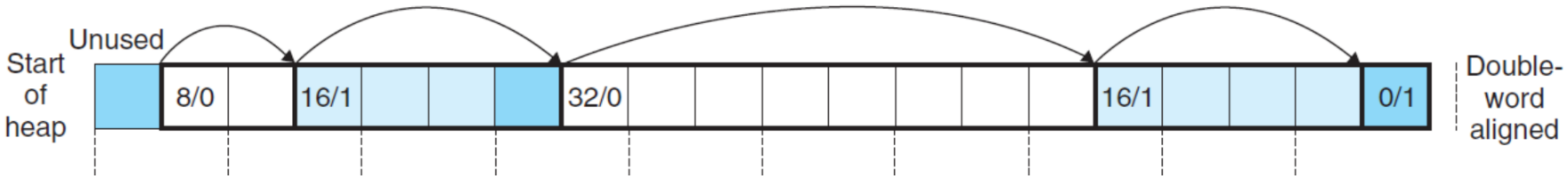
Implicit Free Lists

Any allocator needs some data structure for
- Identifying block boundaries
- Distinguishing between allocated and free blocks

A one-word header encodes the block size and whether the block is allocated or free



Implicit Free Lists



Implicit free list

- Free blocks are linked implicitly by the size field in the header
- Last block: terminating header with size 0 and marked as allocated.
- Simple, but searching for a preceding free block is costly

Placing Allocated Blocks

First fit

- Searches the free list from the beginning and chooses the first free block that fits
- Leaves small splinters towards the beginning of the list. Large free blocks towards the end of the list.

Next fit

- Search the free list from the last allocation point.
- Good chances to find a fit in the remainder of the block
- Suffers from poor memory utilization.

Best fit

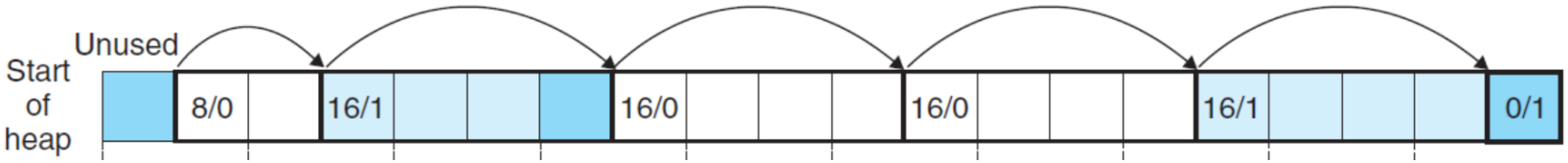
- Searches for a free block with the tightest fit.
- Good memory utilization
- Exhaustive search of the heap

Getting Additional Heap Memory

When the allocator is unable to find a block that fits the request

- Merge adjacent free blocks (coalescing)
- Ask the kernel for additional heap memory by calling sbrk.

Coalescing Free Blocks



False fragmentation

- Adjacent free blocks can serve the request, but individual blocks are too small for the request.

Coalescing merges the adjacent free blocks.

- **Immediate coalescing**: merge free blocks as soon as they are freed.
- **Deferred coalescing**: defer coalescing until later time. (e.g. when some allocation request fails)

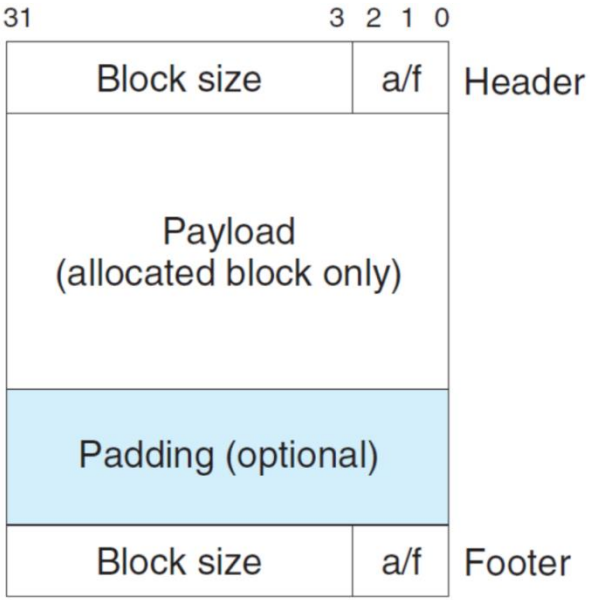
Coalescing with Boundary Tags

Coalescing using the header only

- Coalescing the next free block is straightforward (adding the size to the current block will point to the next block)

- Coalescing with the previous free block requires searching the entire free list

Coalescing with Boundary Tags



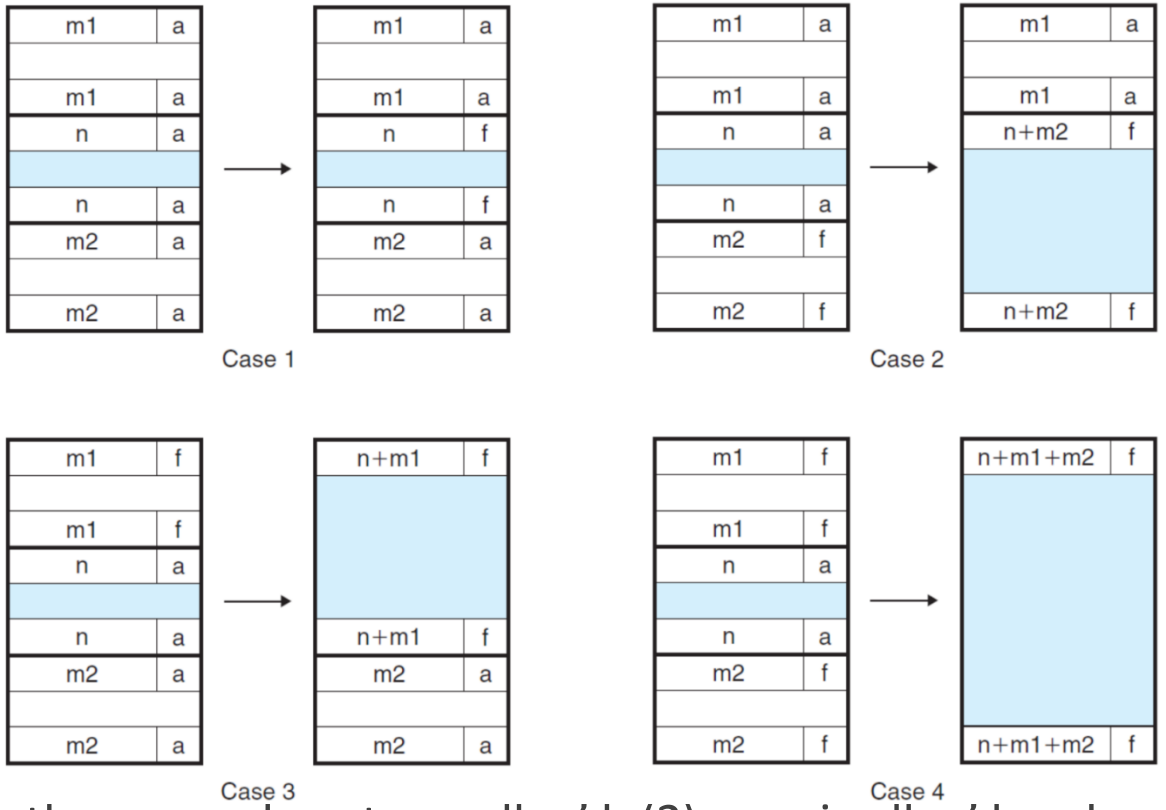
Boundary tag is a footer at the end of the block, where the footer is the replica of the header

Finding the previous block is easy

- Get the size of the previous block from its footer

- If the allocated/free bit for the previous block is encoded at the current block, the footer can be used only for the free blocks

Coalescing with Boundary Tags Example:



4 cases: (1) both prev and next are alloc'd, (2) prev is alloc'd and next is free, (3) prev is free and next is alloc'd, (4) both prev and next are free.

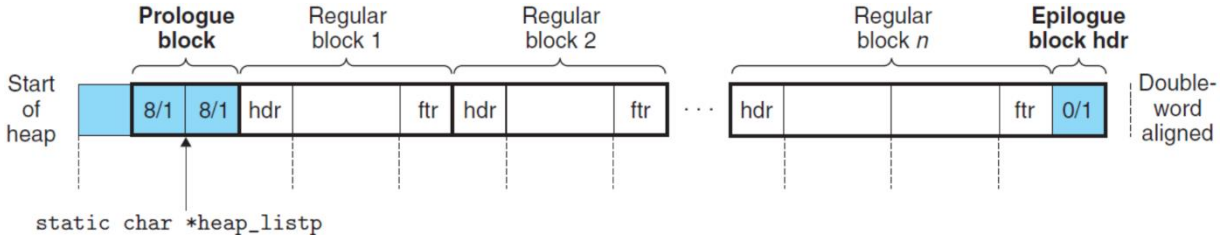
A Simple Allocator

```
/*
 * mem_init - Initialize the memory system model
 */
void mem_init(void)
{
    mem_heap = (char *)Malloc(MAX_HEAP);
    mem_brk = (char *)mem_heap;
    mem_max_addr = (char *)(mem_heap + MAX_HEAP);
}

/*
 * mem_sbrk - Simple model of the sbrk function. Extends the heap
 *    by incr bytes and returns the start address of the new area. In
 *    this model, the heap cannot be shrunk.
 */
void *mem_sbrk(int incr)
{
    char *old_brk = mem_brk;

    if ( (incr < 0) || ((mem_brk + incr) > mem_max_addr)) {
        errno = ENOMEM;
        fprintf(stderr, "ERROR: mem_sbrk failed. Ran out of memory...\n");
        return (void *)-1;
    }
    mem_brk += incr;
    return (void *)old_brk;
}
```

A Simple Allocator (cont)

```
1   /* Basic constants and macros */
2   #define WSIZE       4       /* Word and header/footer size (bytes) */
3   #define DSIZE       8       /* Double word size (bytes) */
4   #define CHUNKSIZE  (1<<12)  /* Extend heap by this amount (bytes) */
5
6   #define MAX(x, y) ((x) > (y)? (x) : (y))
7
8   /* Pack a size and allocated bit into a word */
9   #define PACK(size, alloc)  ((size) | (alloc))
10
11  /* Read and write a word at address p */
12  #define GET(p)       (*(unsigned int *)(p))
13  #define PUT(p, val)  (*(unsigned int *)(p) = (val))
14
15  /* Read the size and allocated fields from address p */
16  #define GET_SIZE(p)  (GET(p) & ~0x7)
17  #define GET_ALLOC(p) (GET(p) & 0x1)
18
19  /* Given block ptr bp, compute address of its header and footer */
20  #define HDRP(bp)       ((char *)(bp) - WSIZE)
21  #define FTRP(bp)       ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)
22
23  /* Given block ptr bp, compute address of next and previous blocks */
24  #define NEXT_BLKP(bp)  ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
25  #define PREV_BLKP(bp)  ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))
```

A Simple Allocator (cont)




```
int mm_init(void)
{
    /* Create the initial empty heap */
    if ((heap_listp = mem_sbrk(4*WSIZE)) == (void *)-1)
        return -1;
    PUT(heap_listp, 0);                          /* Alignment padding */
    PUT(heap_listp + (1*WSIZE), PACK(DSIZE, 1)); /* Prologue header */
    PUT(heap_listp + (2*WSIZE), PACK(DSIZE, 1)); /* Prologue footer */
    PUT(heap_listp + (3*WSIZE), PACK(0, 1));     /* Epilogue header */
    heap_listp += (2*WSIZE);

    /* Extend the empty heap with a free block of CHUNKSIZE bytes */
    if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
        return -1;
    return 0;
}
```

A Simple Allocator (cont)

```
static void *extend_heap(size_t words)
{
    char *bp;
    size_t size;

    /* Allocate an even number of words to maintain alignment */
    size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
    if ((long)(bp = mem_sbrk(size)) == -1)
        return NULL;

    /* Initialize free block header/footer and the epilogue header */
    PUT(HDRP(bp), PACK(size, 0));         /* Free block header */
    PUT(FTRP(bp), PACK(size, 0));         /* Free block footer */
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); /* New epilogue header */

    /* Coalesce if the previous block was free */
    return coalesce(bp);
}
```

A Simple Allocator (cont)

```
void mm_free(void *bp)
{
    size_t size = GET_SIZE(HDRP(bp));

    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    coalesce(bp);
}

static void *coalesce(void *bp)
{
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));

    if (prev_alloc && next_alloc) {          /* Case 1 */
        return bp;
    }
```

A Simple Allocator (cont)

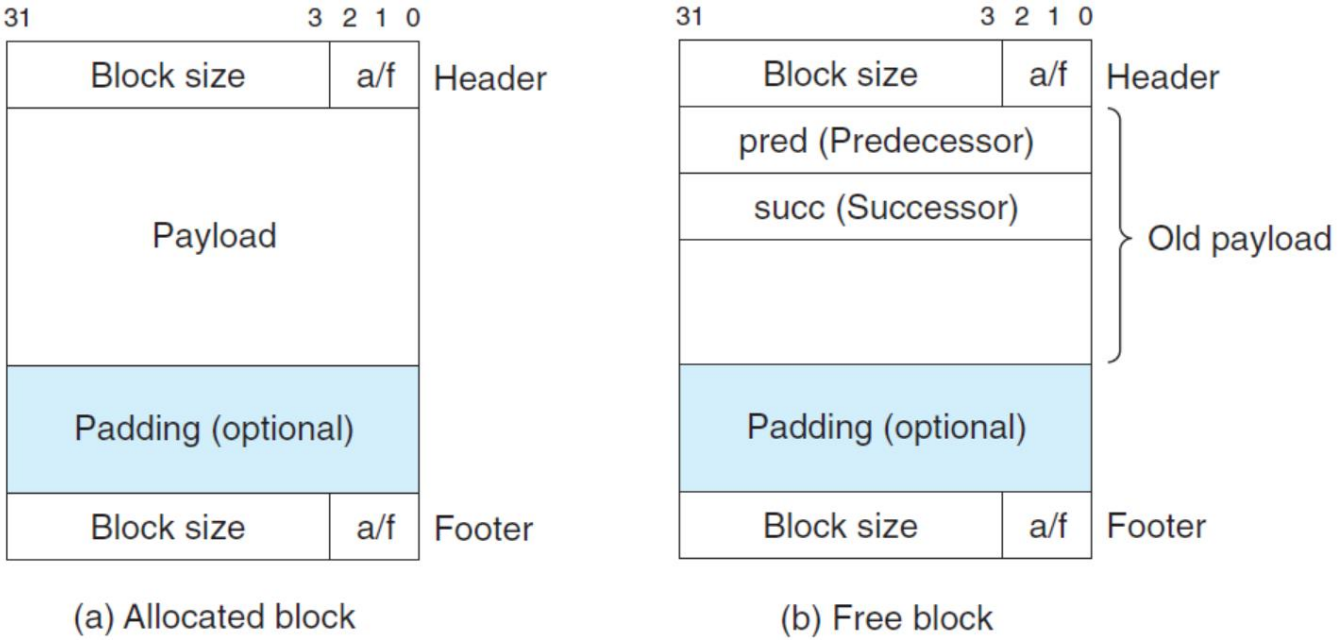
```
    else if (prev_alloc && !next_alloc) {      /* Case 2 */
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size,0));
    }

    else if (!prev_alloc && next_alloc) {      /* Case 3 */
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }

    else {                                     /* Case 4 */
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
            GET_SIZE(FTRP(NEXT_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }
    return bp;
}
```

A Simple Allocator (cont)

```
void *mm_malloc(size_t size)
{
    size_t asize;      /* Adjusted block size */
    size_t extendsize; /* Amount to extend heap if no fit */
    char *bp;

    /* Ignore spurious requests */
    if (size == 0)
        return NULL;

    /* Adjust block size to include overhead and alignment reqs. */
    if (size <= DSIZE)
        asize = 2*DSIZE;
    else
        asize = DSIZE * ((size + (DSIZE) + (DSIZE-1)) / DSIZE);     }
```

```
/* Search the free list for a fit */
if ((bp = find_fit(asize)) != NULL) {
    place(bp, asize);
    return bp;
}

/* No fit found. Get more memory and place the block */
extendsize = MAX(asize,CHUNKSIZE);
if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
    return NULL;
place(bp, asize);
return bp;
```

Explicit Free List



For the free blocks, add pred and next link to the previous and the next free blocks.

Reducing Allocation Time

- Segregated Free Lists
  - Free blocks organized into different *equivalence classes* based on size
  - Numerous approaches differing in several aspects
    - How size classes are defined
    - When they coalesce
    - When they request additional heap storage
    - If blocks are split
    - Others…
  - Two popular approaches
    - Simple Segregated Storage
    - Segregated Fits

Simple Segregated Storage

- Block sizes are powers of 2: [4, 8, 16, 32, 64, 128, 256, etc]
- Separate free lists for each size
- Blocks are not split. Allocations come from the next larger size based on request:
  - Request for 58 bytes gets a block from the list of 64 byte blocks (which can handle 33-64 byte requests)
  - Request for 130 bytes comes from list of 256 byte blocks
- When no blocks of right size available:
  - More heap space requested
  - Divided into a new list of the correct size

Simple Segregated Storage

- Pluses:
  - Fast constant time allocation
  - No allocation flag or header/footer needed (no coalescing)
  - Only need Single linked list (allocations all from front of each list)
- Minuses:
  - Susceptible to both internal and external fragmentation

Segregated Fits

- Maintain several free lists based on size classes
- Blocks in a specific size class vary in size (instead of all the same size)
- Do a ‘first fit’ allocation. If none fit from the list, move to next larger size equivalence class
- Blocks are coalesced when freed and added to the correct free list

Segregated Fits

- Pluses:
  - Search times reduced due to searching only appropriate size classes
  - Fast and efficient
  - Memory efficient since it approximates best fit across entire heap

Buddy Systems

Special case of Segregated Fits

All blocks are sized as powers of 2 (2^2, 2^3, 2^4, 2^5, 2^6, etc.)

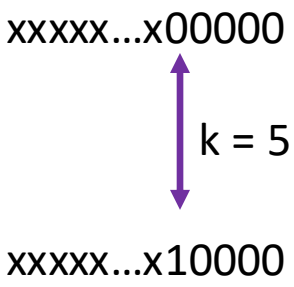
Process:

- Heap starts as 1 block of size 2^m
- **Allocation:**
  - Block requests rounded up to power of 2 (2^k)
  - Find Free blocks on list with block sizes as close as possible (2^j)
    - k <= j <= m
    - if j == k, done
    - Otherwise: recursively split block until j == k. Add unallocated 'buddy' to the correct size free list.
- Free:
  - As long as matching 'buddy' is free, coalesce blocks
  - Add final coalesced block to the correct free list

Buddy Systems

Addresses of allocated blocks and 'buddies' differe by 1 bit at position (k-1):

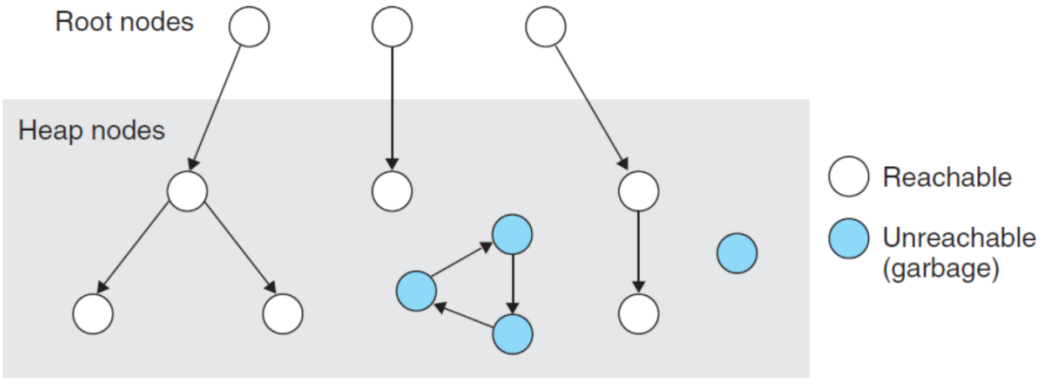
- $32 = 2^5$



This makes coallescing easy and no header/footer words are required

Garbage Collection

```
void CreateGarbage()
{
    int *p = (int*)
          malloc(100);
    return;
}
```



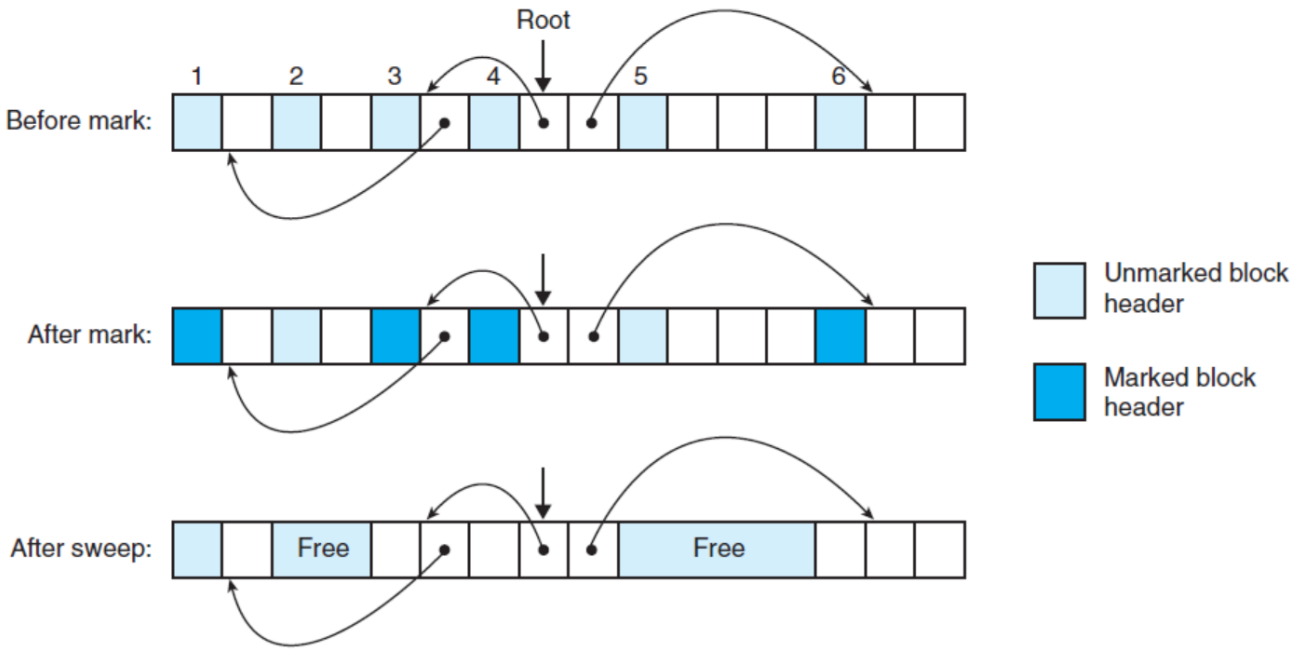
Garbage

- Any variable not reachable from your program

Reachability Graph

- Nodes are variables
- If a pointer variable v_i is pointing to another variable v_j, there is an edge $v_i \rightarrow v_j$
- A variable v_i is reachable if there is a path to v_i from any root variables (live variables not in the heap)

Garbage Collection



Mark&Sweep Garbage Collector

- Mark phase: mark all variables reachable from any root variables
- Sweep phase: free the variables not marked during the mark phase.

Questions?