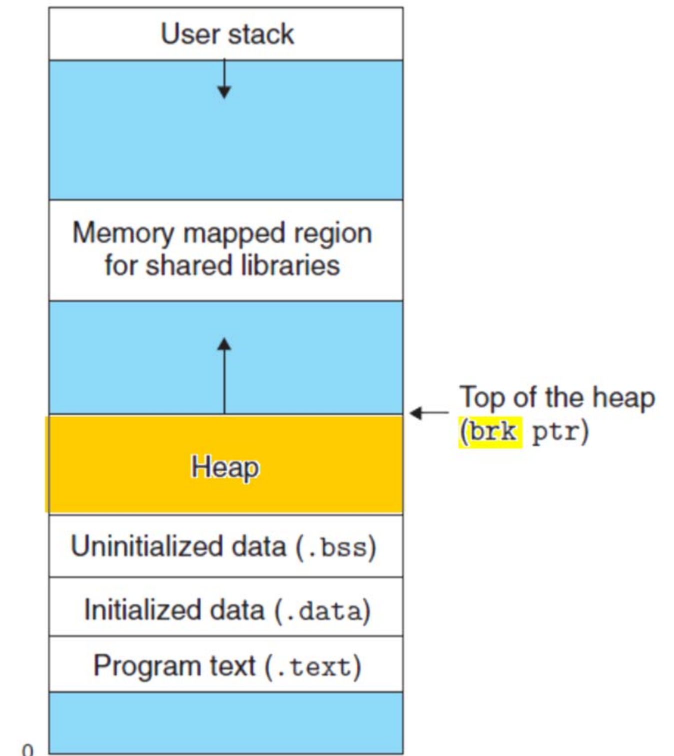
# CSE320 System Fundamentals II
# Dynamic Memory Allocation

YoungMin Kwon

SUNY Korea
The State University of New York

# 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
  - Garbage collectors

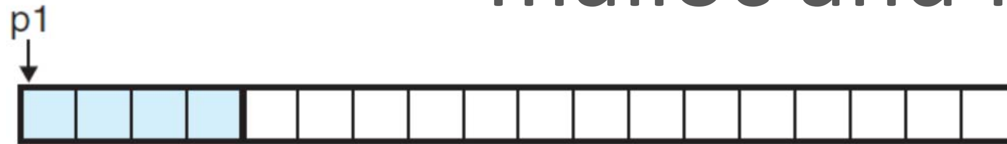


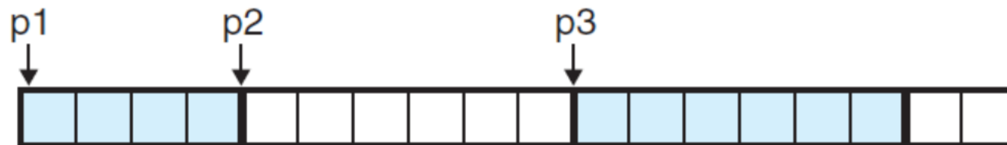
# malloc and free

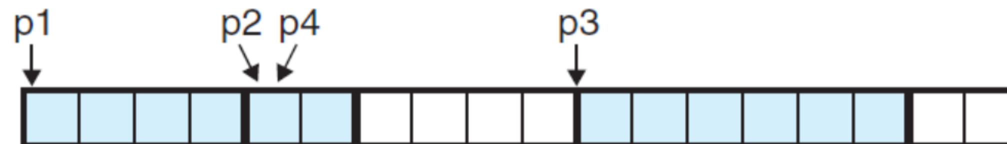
- void *malloc(size_t size) allocates at least the request size bytes
  - calloc works like malloc but initializes the memory to 0 in addition
- void free(void *ptr) frees the allocated memory
- void *sbrk(intptr_t incr) increases brk by incr and returns the old brk.

SUNY Korea

# malloc and free






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

Dark blue area is a padding for the alignment

SUNY Korea

# 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))` in (e) of prev. page

SUNY Korea

# Implementation Issues

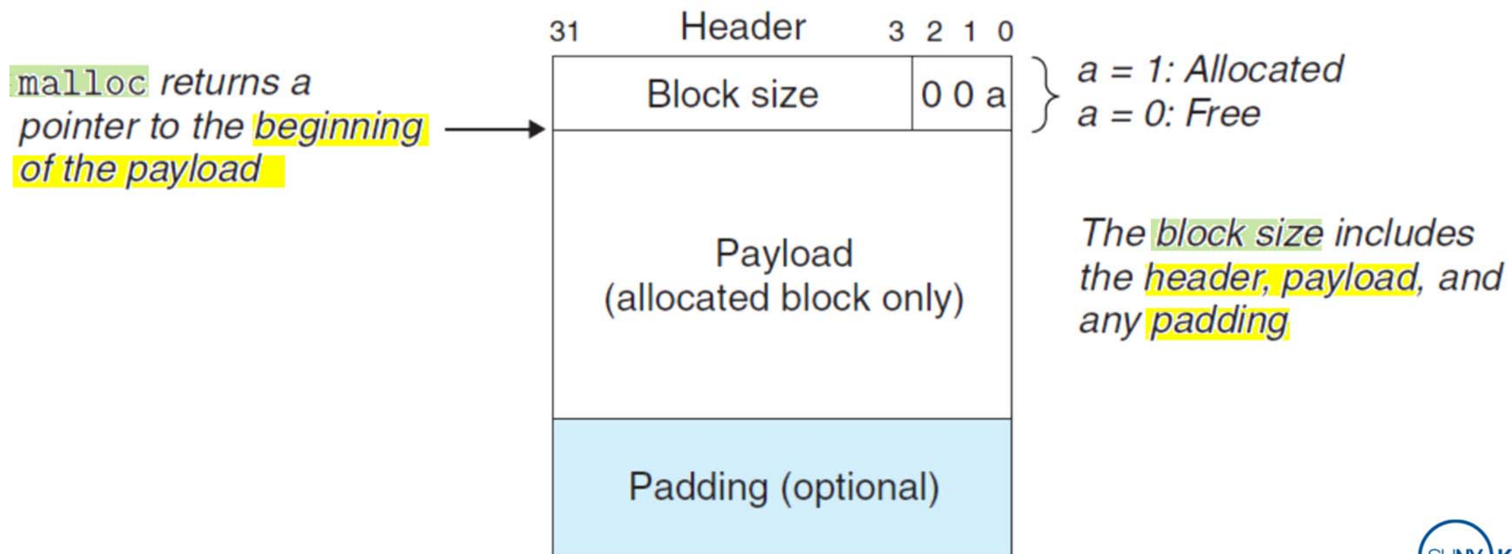
- Free block organization
  - How to keep track of free blocks?
- Placement
  - How to choose a free block for a request?
- Splitting
  - When a part of a free block is allocated, what do we do for the remaining free blocks?
- Coalescing
  - What do we do with a block that is just freed?

SUNY Korea

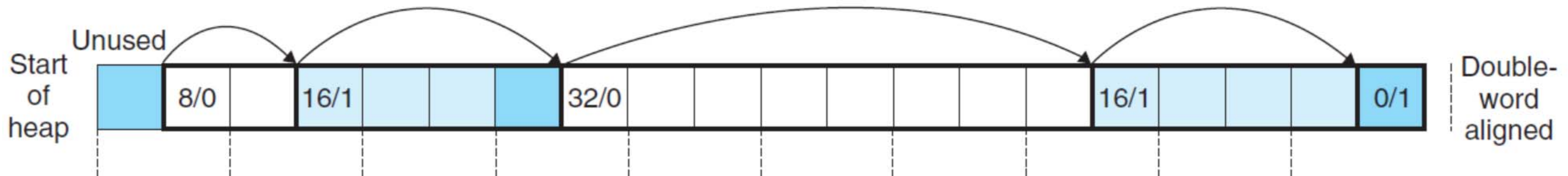
# Implicit Free Lists

- 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

SUNY Korea

# Placing Allocated Blocks

- First fit
  - Search the free list from the beginning and choose the first free block that fits
  - Leave 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 the memory utilization.
- Best fit
  - Searches for a free block with the tightest fit.
  - Good memory utilization
  - Exhaustive search of the heap

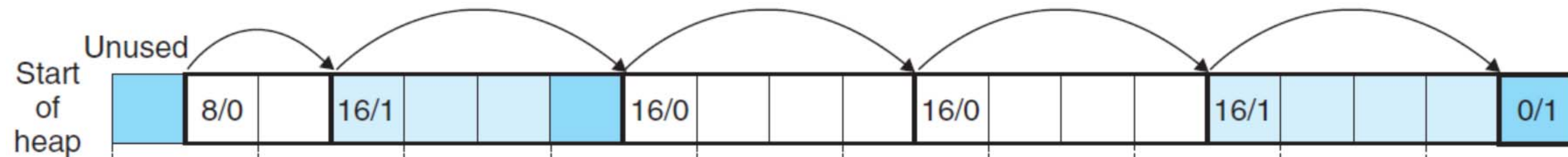
SUNY Korea

# 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.

SUNY Korea

# 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)

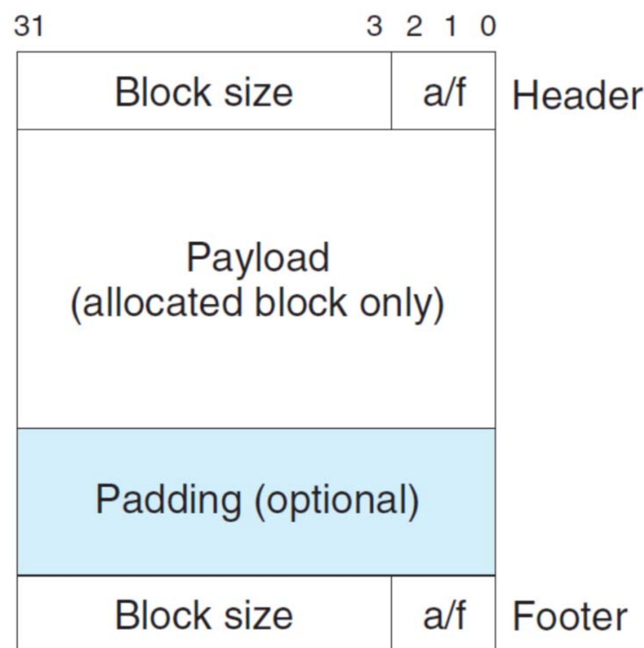
SUNY Korea

# 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

SUNY Korea

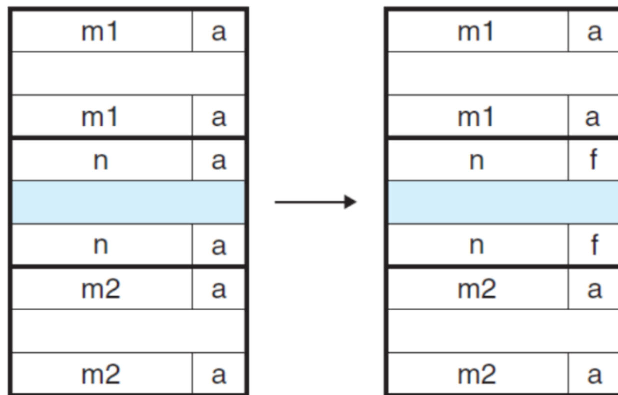
# Coalescing with Boundary Tags



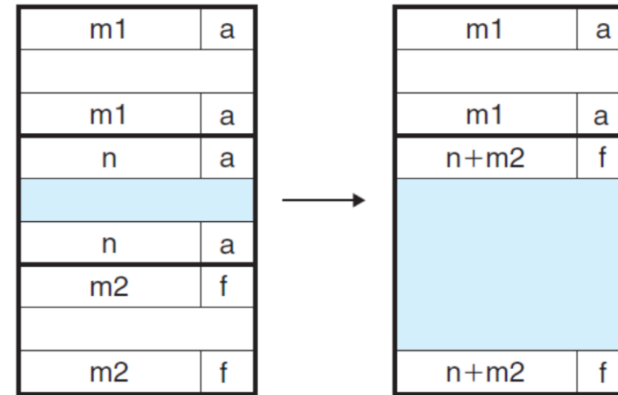

- Boundary tag is a footer at the end of the block
  - The footer is the replica of the header
- Finding the previous block is easy
  - Get the size of the previous block from its footer

- 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

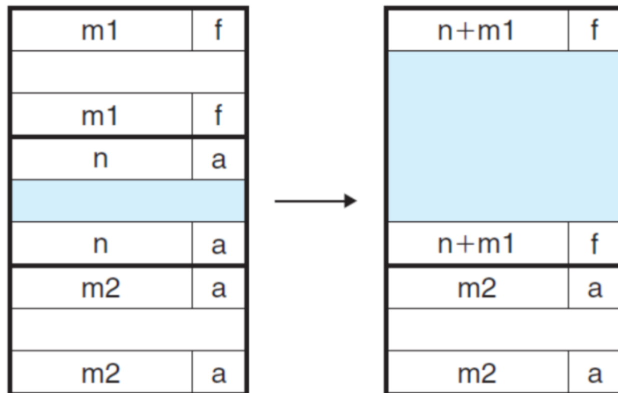
SUNY Korea



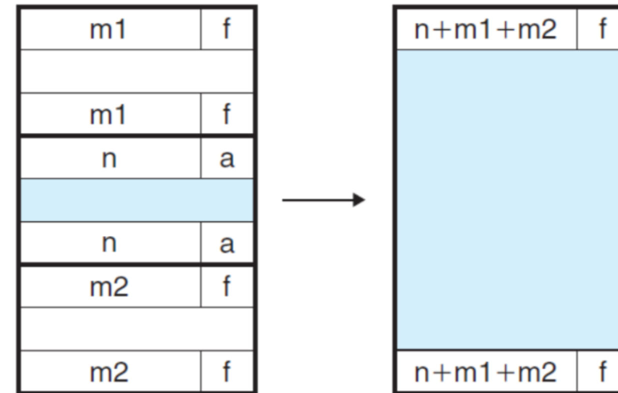

Case 1




Case 2




Case 3




Case 4

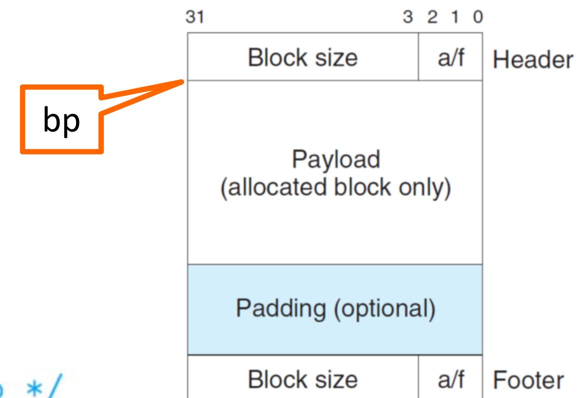
4 cases: (1) both prev and next are alloc'd, (2) prev is alloc'd and next is free, (3) prev is free and next is alloc'd, (4) both prev and next are free.

# 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;
}
```

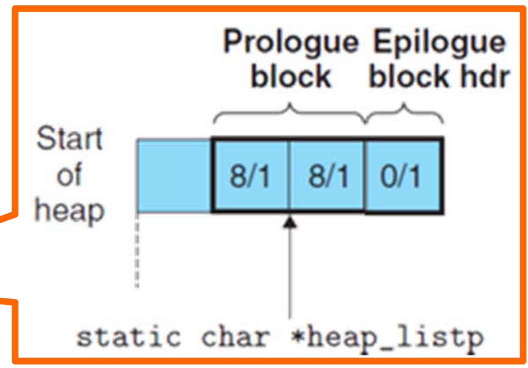
SUNY Korea

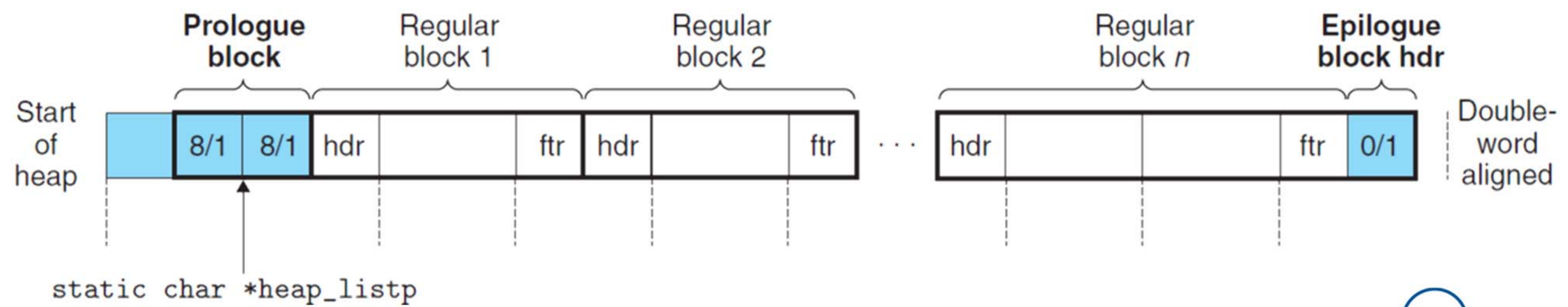
```
1   /* Basic constants and macros */
2   #define WSIZE       4       /* Word and header/footer size (bytes) */
3   #define DSIZE       8       /* Double word size (bytes) */
4   #define CHUNKSIZE  (1<<12)  /* Extend heap by this amount (bytes) */
5
6   #define MAX(x, y) ((x) > (y)? (x) : (y))
7
8   /* 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)))
```




SUNY Korea

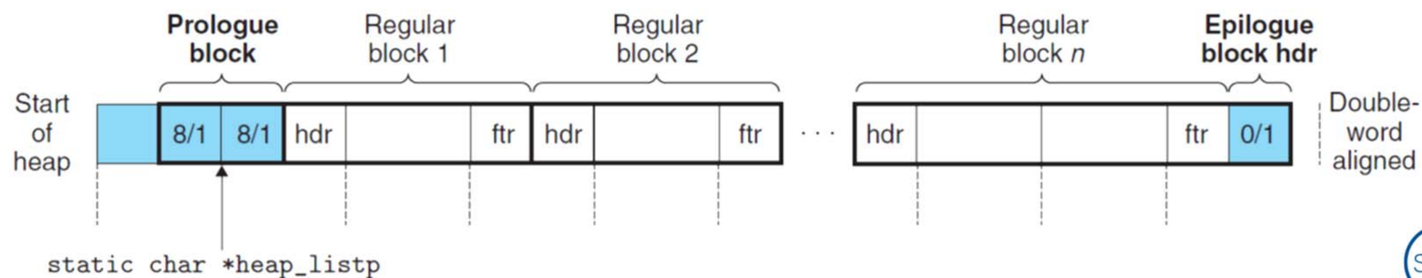
```c
int mm_init(void)
{
    /* Create the initial empty heap */
    if ((heap_listp = mem_sbrk(4*WSIZE)) == (void *)-1)
        return -1;
    PUT(heap_listp, 0);                          /* Alignment padding */
    PUT(heap_listp + (1*WSIZE), PACK(DSIZE, 1)); /* Prologue header */
    PUT(heap_listp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



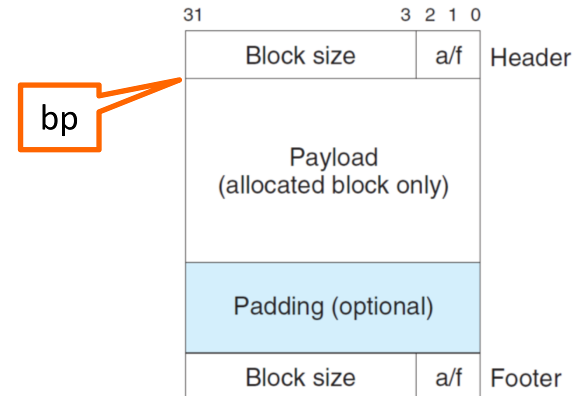

SUNY Korea

```c
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);
}
```




SUNY Korea

```
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;
    }
```

SUNY Korea

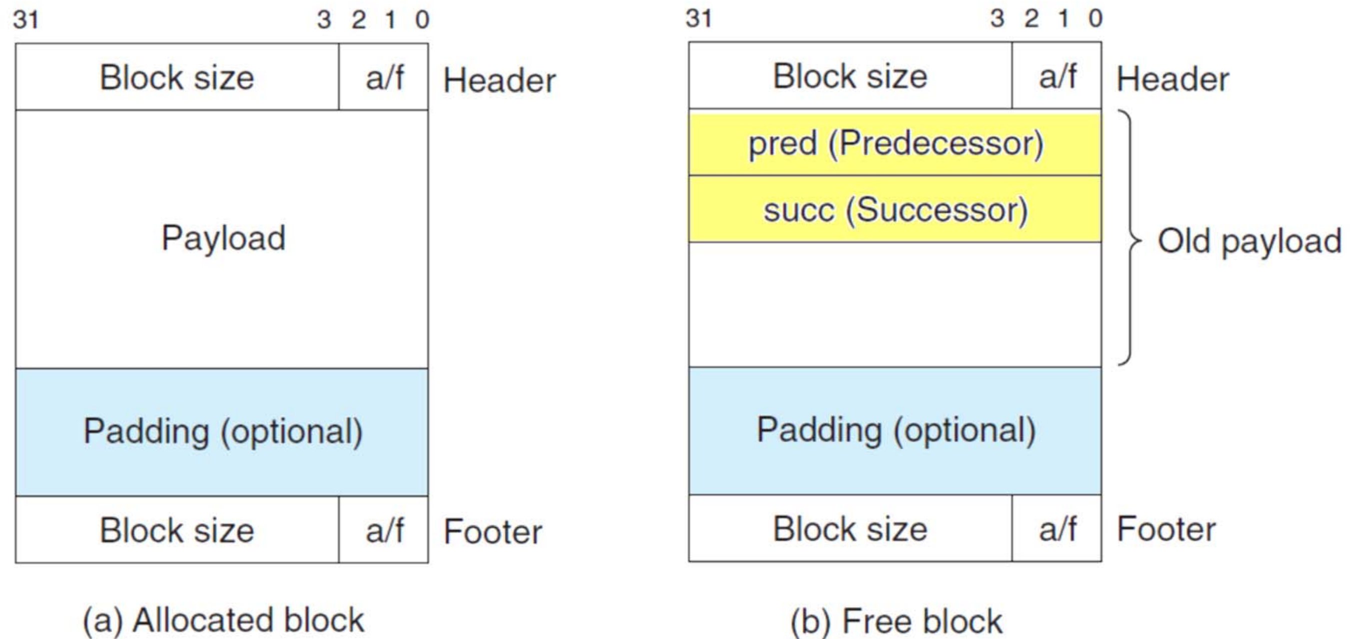
```
    else if (prev_alloc && !next_alloc) {      /* Case 2 */
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size,0));
    }

    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;
}
```

SUNY Korea
The State University of New York

```c
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;
}
```

SUNY Korea
The State University of New York

# Explicit Free List

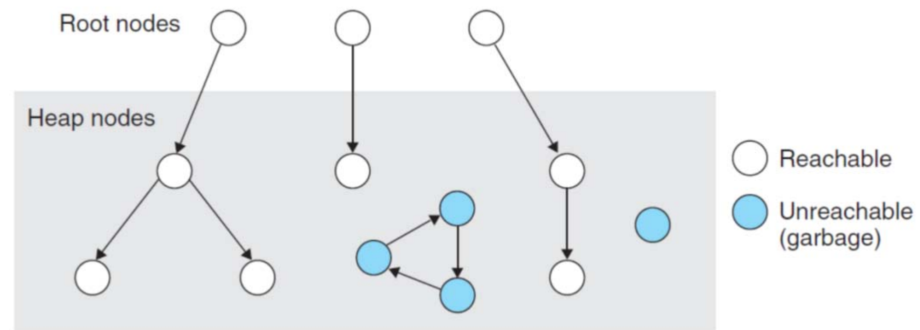



- For the free blocks, add pred and succ link to the previous and the next free blocks.

SUNY Korea

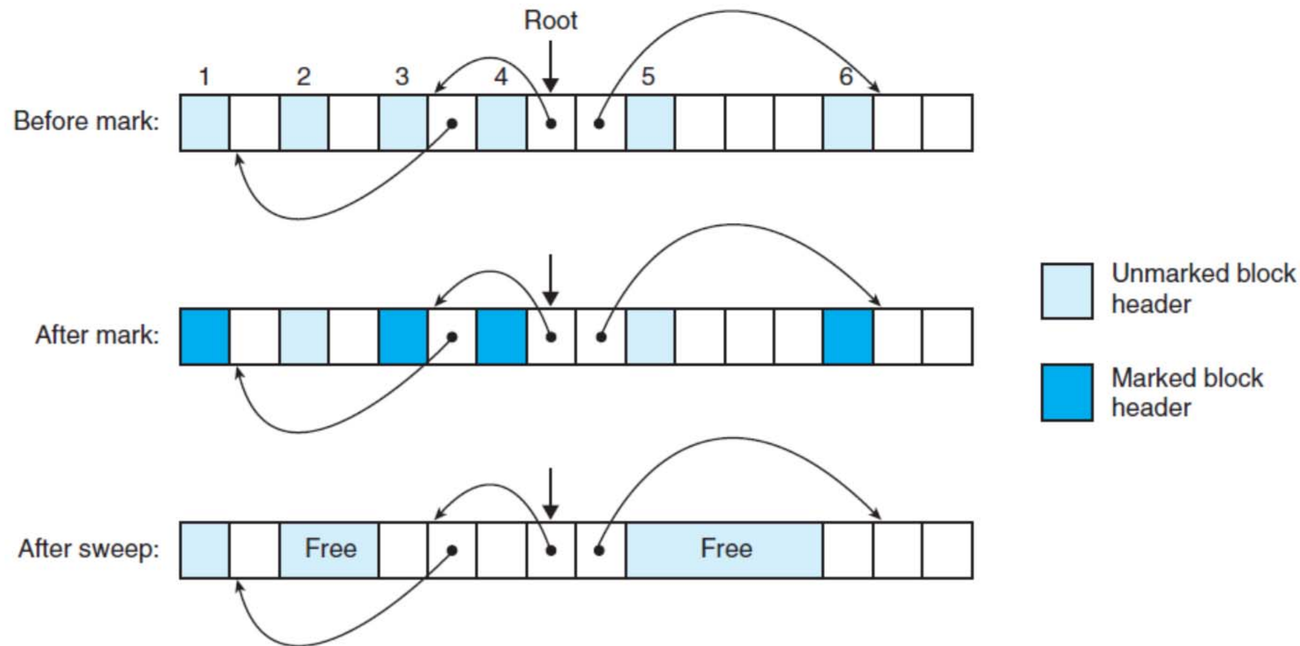
# Garbage Collection

```
void make_garbage()
{
    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.

SUNY Korea

# Programming Assignment 5

- The program in the next pages is an implementation of malloc and free functions on a simulated Heap

- Implement the TODO lines and check your results against the expected ones

- Due date TBD

SUNY Korea

```c
int main() {
    extern void unit_test();
    unit_test();
}

// unit test
void unit_test() {
    heap_mgr_t *hmgr = make_heap_mgr_default();
…
}

// memory
typedef struct memory {
    char *start;
    char *end;
    char *brk;

    void *(*sbrk)(struct memory *self, size_t incr);
} memory_t;

static void *sbrk(memory_t *self, size_t incr) {
    char *oldbrk = self->brk;
    ON_FALSE_EXIT(incr >= 0 && self->brk + incr < self->end, "out of memory");
    self->brk += incr;
    return oldbrk;
}
```

SUNY Korea

```c
typedef unsigned long block_hdr_t;

typedef struct block {
    block_hdr_t prev_footer;
    block_hdr_t header;
    char      payload;        //where the payload begins
} block_t;

typedef struct block_mgr {
    //functions related to header
    int         (*hdr_inuse)(block_hdr_t hdr);
    size_t      (*hdr_size)(block_hdr_t hdr);
    block_hdr_t (*hdr_make)(size_t size, int inuse);
    block_hdr_t (*hdr_make_epilog)();

    //functions related to block
    int         (*inuse)(block_t *blk);
    size_t      (*size)(block_t *blk);
    void        (*set_header)(block_t *blk, size_t size, int inuse);
    block_t    *(*next)(block_t *blk);
    block_t    *(*prev)(block_t *blk);
} block_mgr_t;
```

SUNY Korea

```c
// block related functions
//
static int block_inuse(block_t *blk) {
    return hdr_inuse(blk->header);
}

static size_t block_size(block_t *blk) {
    return hdr_size(blk->header);
}

static void block_set_header(block_t *blk, size_t size, int inuse) {
    inuse = !!inuse;    //make it either 0 or 1
    //TODO: update blk's header and its next block's prev_footer
}

static block_t *block_next(block_t *blk) {
    char *ptr = (char*)blk;
    //TODO: return the next block
}

static block_t *block_prev(block_t *blk) {
    char *ptr = (char*)blk;
    //TODO: return the previous block
}
```

SUNY Korea

```c
typedef struct heap_mgr {
    void     (*init_heap)(struct heap_mgr *self);
    block_t *(*extend_heap)(struct heap_mgr *self, size_t size);
    block_t *(*coalesce)(struct heap_mgr *self, block_t *curr);
    block_t *(*find_free_block)(struct heap_mgr *self, size_t size);
    void     (*place)(struct heap_mgr *self, block_t *curr, size_t size);
    size_t   (*mem_aligned_size)(struct heap_mgr *self, size_t size);

    //equivalent to malloc of stdlib
    void    *(*malloc)(struct heap_mgr *self, size_t size);
    //equivalent to free of stdlib
    void     (*free)(struct heap_mgr *self, void *ptr);

    memory_t *heap;
    block_mgr_t *bmgr;      //block manager
    block_t *blist_head;    //block list head
    size_t mem_align;       //mem align boundary
    size_t heap_size;       //heap size
    size_t brk_min_incr;    //minimum brk increase size
} heap_mgr_t;
```

SUNY Korea

```c
static block_t *extend_heap(heap_mgr_t *self, size_t size) {
    block_mgr_t *bmgr = self->bmgr;
    memory_t *heap = self->heap;

    //adjust size
    if(size < self->brk_min_incr)
        size = self->brk_min_incr;
    size = self->mem_aligned_size(self, size);

    //find and update the last block

    //TODO: increase brk and get the oldbrk
    //hint: use heap->sbrk

    //TODO: from the oldbrk, find the last block (its containing block)
    //hint: use containerof
    //hint: oldbrk is the address of the block's payload

    //TODO: set the header of the last block
    //hint: use bmgr->set_header

    //the new last block
    //TODO: get the next block of the last block

    //TODO: copy epilog to the header of the block

    //TODO: coalese the last block and return the result
    //hint: use self->coalesce
}
```

SUNY Korea

```c
static block_t *coalesce(heap_mgr_t *self, block_t *curr) {
    block_mgr_t *bmgr = self->bmgr;
    block_t *prev = bmgr->prev(curr);
    block_t *next = bmgr->next(curr);

    //TODO: handle the four cases
    //hint: compute the new size and update the header
    //hint: use bmgr->set_header
    if(bmgr->inuse(prev) &&
       bmgr->inuse(next)) {

    }
    else if( bmgr->inuse(prev) &&
            !bmgr->inuse(next)) {

    }
    else if(!bmgr->inuse(prev) &&
             bmgr->inuse(next)) {

    }
    else {

    }
}
```

SUNY Korea

```c
static block_t *find_free_block(heap_mgr_t *self, size_t size) {
    block_mgr_t *bmgr = self->bmgr;
    block_hdr_t epilog = bmgr->hdr_make_epilog();
    block_t *pos;

    //TODO: find and return a free block whose size is at least size param
    //hint: step through the linked list of blocks using for loop
    //hint: start pos from self->blist_head
    //hint: stop when the pos' header is equal to epilog
    //hint: use bmgr->next to find the next block

    return NULL;
}
```

SUNY Korea

```c
//place size bytes of allocated block on curr
static void place(heap_mgr_t *self, block_t *curr, size_t size) {
    block_mgr_t *bmgr = self->bmgr;
    size_t next_size;

    //TODO: compute next_size.
    //      next_size is the size of the remaining block after split

    //TODO: set the header of the free block

    if(next_size >= 2*sizeof(block_hdr_t)) {
        //split
        //TODO: set the header of the remaining block

    }
    else {
        //no split
        ON_FALSE_EXIT(next_size == 0, "unexpected next size");
    }
}
```

SUNY Korea

```c
//equivalent to malloc of stdlib
static void *mem_alloc(heap_mgr_t *self, size_t size) {
    block_mgr_t *bmgr = self->bmgr;
    block_t *curr;

    size = self->mem_aligned_size(self, size + 2*sizeof(block_hdr_t));

    //TODO: find a free block with enough size; extend the heap if not found

    //TODO: place the block found

    //TODO: return the address of the curr's payload
}


//equivalent to free of stdlib
static void mem_free(heap_mgr_t *self, void *ptr) {
    block_mgr_t *bmgr = self->bmgr;
    block_t *curr;

    //TODO: find the block_t address of ptr
    //hint: use containerof

    //TODO: update the inuse field of curr

    //TODO: coalesce
}
```

SUNY Korea

- Expected result

```
$ ./a.out
Malloc ptr[0] = 0x7f430a272030
Malloc ptr[1] = 0x7f430a272430
Malloc ptr[2] = 0x7f430a272830
Malloc ptr[3] = 0x7f430a272c30
Malloc ptr[4] = 0x7f430a273030
Malloc ptr[5] = 0x7f430a273430
Malloc ptr[6] = 0x7f430a273830
Malloc ptr[7] = 0x7f430a273c30
Malloc ptr[8] = 0x7f430a274030
Malloc ptr[9] = 0x7f430a274430
Free ptr[1] = 0x7f430a272430
Free ptr[3] = 0x7f430a272c30
Free ptr[5] = 0x7f430a273430
Free ptr[7] = 0x7f430a273c30
Free ptr[9] = 0x7f430a274430
p:0x7f430a272430 == ptr[1]:0x7f430a272430
p:0x7f430a274430 == ptr[9]:0x7f430a274430
p:0x7f430a272c30 == ptr[3]:0x7f430a272c30
p:0x7f430a272e40 > ptr[3]:0x7f430a272c30
p:0x7f430a272e40 < ptr[4]:0x7f430a273030
SUCCESS!
```

SUNY Korea