CS356: Discussion #12

Dynamic Memory and Allocation Lab

Marco Paolieri (paolieri@usc.edu)
Illustrations from CS:APP3e textbook

USC University of Southern California

Dynamic Memory Allocation in C

Low-level memory allocation in Linux

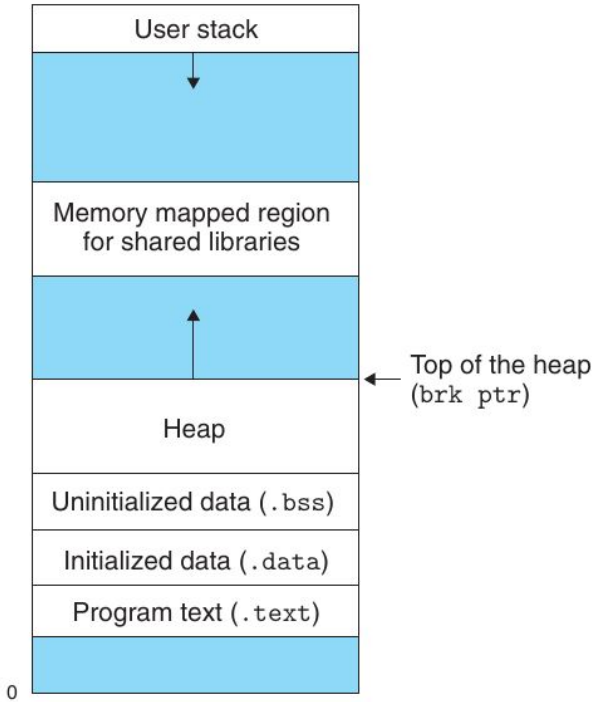
```
void *mmap(void *addr, size_t length,
  int prot, int flags, int fd, off_t off)
int munmap(void *addr, size_t length)
void *sbrk(intptr_t increment)
```

Portable alternatives in `stdlib`

```
void *malloc(size_t size)
void *calloc(size_t nmemb, size_t size)
void *realloc(void *ptr, size_t size)
void free(void *ptr)
```

Check man pages for these functions!

```
$ man malloc
```



Online documentation

NAME

brk, sbrk — change space allocation

SYNOPSIS

UX

```
#include <unistd.h>

int brk(void *addr);

void *sbrk(int incr);
```

DESCRIPTION

The *brk*( ) and *sbrk*( ) functions are used to change the amount of space allocated for the calling process. The change is made by resetting the process' break value and allocating the appropriate amount of space. The amount of allocated space increases as the break value increases. The newly-allocated space is set to 0. However, if the application first decrements and then increments the break value, the contents of the reallocated space are unspecified.

The *brk*( ) function sets the break value to *addr* and changes the allocated space accordingly.

The *sbrk*( ) function adds *incr* bytes to the break value and changes the allocated space accordingly. If *incr* is negative, the amount of allocated space is decreased by *incr* bytes. The current value of the program break is returned by *sbrk*(0).

The behaviour of *brk*( ) and *sbrk*( ) is unspecified if an application also uses any other memory functions (such as *malloc*( ), *mmap*( ), *free*( )). Other functions may use these other memory functions silently.

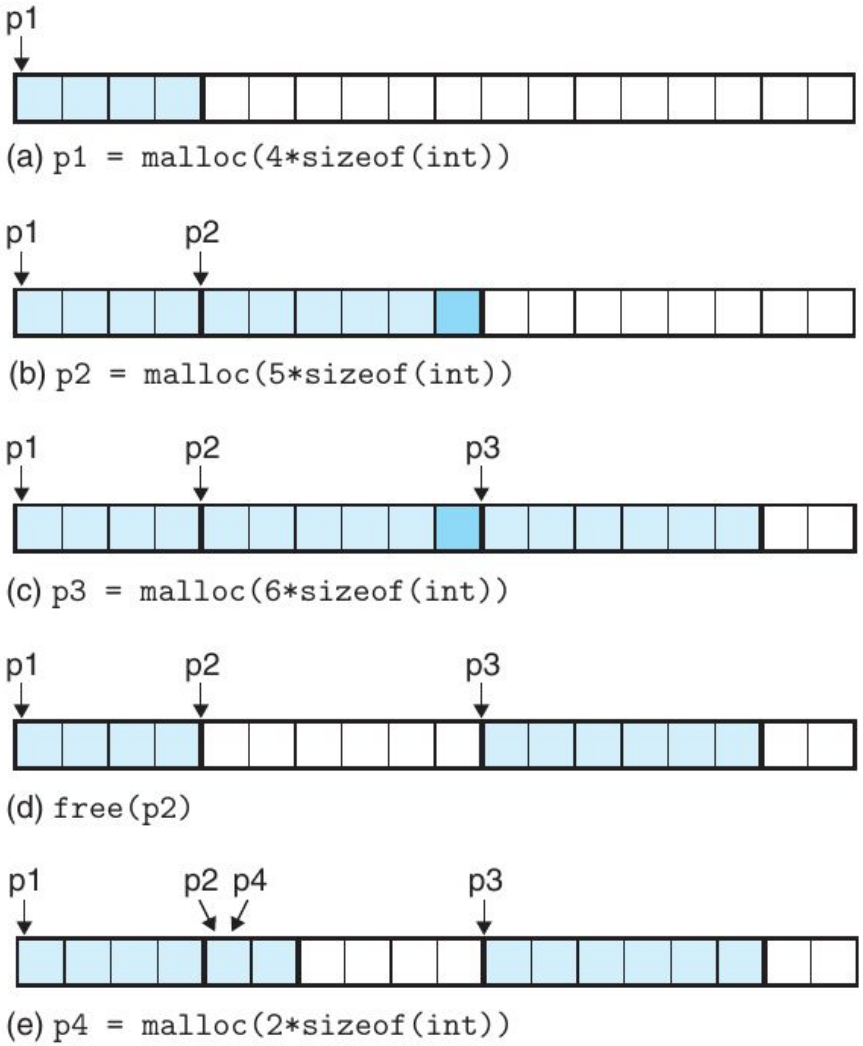
RETURN VALUE

Upon successful completion, *brk*( ) returns 0. Otherwise, it returns –1 and sets *errno* to indicate the error.

Upon successful completion, *sbrk*( ) returns the prior break value. Otherwise, it returns (void *)–1 and sets *errno* to indicate the error.

Source: http://pubs.opengroup.org/onlinepubs/9695969499/toc.pdf#page=92

`malloc` and `free` in action



Each square represents 4 bytes.

`malloc` returns addresses at multiples of 8 bytes (64 bit).

If there is a problem, `malloc` returns `NULL` and sets `errno`.

`malloc` does not initialize to 0, use `calloc` instead.

Using `malloc` and `free`

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

int compare(const void *x, const void *y) {
    int xval = *(int *)x;
    int yval = *(int *)y;

    if (xval < yval)
        return -1;
    else if (xval == yval)
        return 0;
    else
        return 1;
}
```

```c
int main() {
    int count = 0;
    printf("How many numbers to sort? ");
    if (scanf("%d", &count) != 1) {
        fprintf(stderr, "Invalid input\n");
        return 1;
    }
    int *numbers = (int *) malloc(count * sizeof(int));
    printf("Please input %d numbers: ", count);
    for (int i = 0; i < count; i++) {
        if (scanf("%d", &numbers[i]) != 1) {
            fprintf(stderr, "Invalid input\n");
            return 1;
        }
    }
    qsort(numbers, count, sizeof(int), compare);
    printf("Sorted numbers:");
    for (int i = 0; i < count; i++) {
        printf(" %d", numbers[i]);
    }
    printf("\n");
    free(numbers);
    return 0;
}
```

```
$ gcc sort.c -o sort
$ ./sort
How many numbers to sort? 4
Please input 4 numbers: 2 3 1 -1
Sorted numbers: -1 1 2 3
```

Memory-Related Bugs

```c
/* Return y = Ax */
int *matvec(int **A, int *x, int n) {
    int i, j;
    int *y = (int *)malloc(n * sizeof(int));
    /* should set y[i] = 0 (or use calloc) */
    for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
          y[i] += A[i][j] * x[j];
      }
    }
    return y;
}
```

```c
void leak(int n) {
    int *x = (int *)malloc(n * sizeof(int));
    return;
    /* x is garbage at this point */
}
```

```c
int *stackref() {
    int val;
    return &val;  /* val is a local variable */
}
```

```c
void buffer_overflow() {
    char buf[64];
    gets(buf);  /* use fgets instead */
    return;
}
```

```c
void bad_pointer() {
    int val;
    scanf("%d", val);  /* use &val */
}
```

```c
int *search(int *p, int val) {
    while (*p && *p != val)
      p += sizeof(int);  /* should be p++ */
    return p;
}
```

Memory-Related Bugs

```c
/* Create an nxm array */
int **makeArray1(int n, int m) {
    int i;
    /* should be sizeof(int*) */
    int **A = (int **)malloc(n * sizeof(int));
    for (i = 0; i < n; i++) {
      A[i] = (int *)malloc(m * sizeof(int));
    }
    return A;
}

/* Create an nxm array */
int **makeArray2(int n, int m) {
    int i;
    int **A = (int **)malloc(n * sizeof(int *));
    /* should be .. i < n .. */
    for (i = 0; i <= n; i++) {
        A[i] = (int *)malloc(m * sizeof(int));
    }
    return A;
}
```

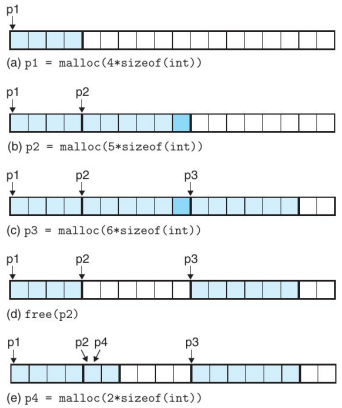
```c
int *binheapDelete(int **binheap, int *size) {
    int *packet = binheap[0];
    binheap[0] = binheap[*size - 1];
    *size--; /* This should be (*size)-- */
    heapify(binheap, *size, 0);
    return(packet);
}
```

```c
int *heapref(int n, int m) {
    int i;
    int *x, *y;
    x = (int *)malloc(n * sizeof(int));
    /* ... */
    /* Other calls to malloc and free here */
    free(x);
    y = (int *)malloc(m * sizeof(int));
    for (i = 0; i < m; i++) {
      y[i] = x[i]++; /* x[i] in freed block */
    }
    return y;
}
```

Explicit Heap Allocators

Explicit allocators like `malloc`

- Must handle arbitrary sequences of allocate/free requests
- Must respond immediately (no buffering of requests)
- Helper data structures must be stored in the heap itself
- Payloads must be aligned to 8-bytes boundaries
- **Allocated blocks cannot be moved/modified**



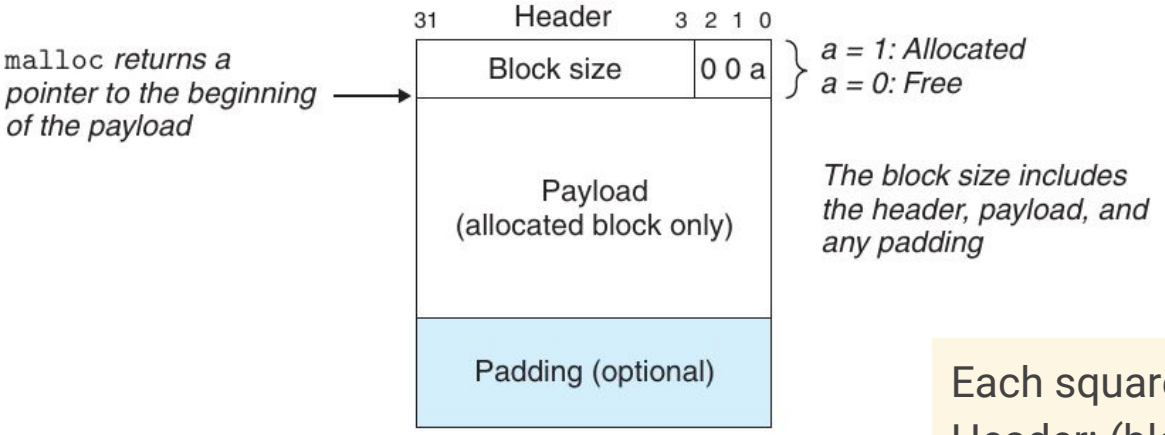
Goal 1. Maximize throughput: (# completed requests) / second

- Simple to implement `malloc` with running time O(# free blocks) and `free` with running time O(1)

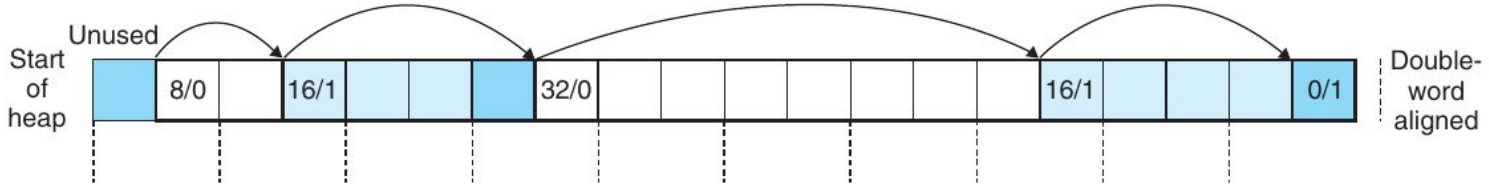
Goal 2. Maximize peak utilization: max{ allocated(t) : t ⩽ T} / heapsize(T)

- If the heap can shrink, take max{ heapsize(t) : t ⩽ T}
- Problem: **fragmentation**, internal (e.g., larger block allocated for alignment) or external (free space between allocated blocks)
- Severity of external fragmentation depends also on **future requests**

Implementation: Implicit Free Lists



Each square represents 4 bytes
Header: (block size) / (allocated)



- Block size includes header/padding, always a multiple of 8 bytes.
- Can scan the list using headers but **O(# blocks)**, not O(# free blocks)
- Special terminating header: zero size, allocated bit set (will not be merged)
- With 1-word header and 2-word alignment, **minimum block size is 2 words**

Exercise

Assume:

- 8-byte alignment
- Block sizes multiples of 8 bytes
- Implicit free list with 4-byte header (format from previous slide)

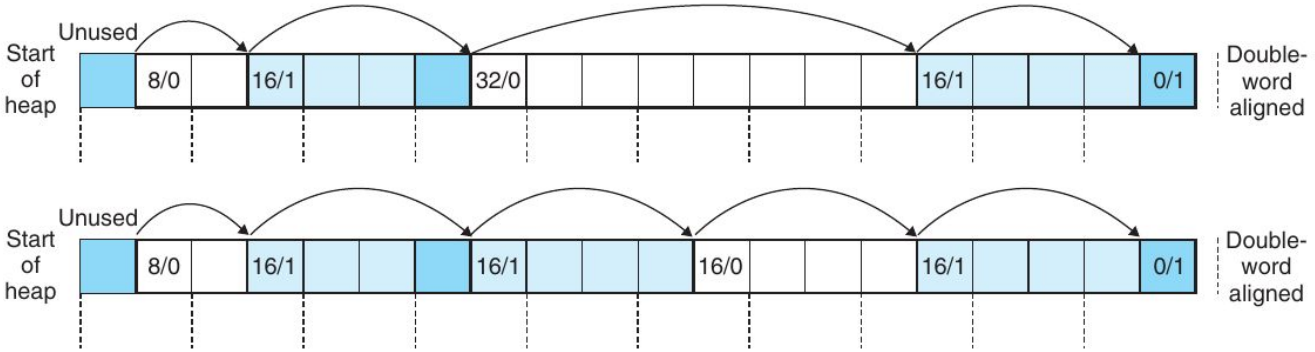
Block size (in bytes) and block header (in hex) for blocks allocated by the following sequence:

```
malloc(1)  malloc(5)  malloc(12)  malloc(13)
```

- **malloc(1):** 8 bytes, block header 0x00000009 == 0...01001
- **malloc(5):** 16 bytes, block header 0x00000011 == 0...10001
- **malloc(12):** 16 bytes, block header 0x00000011 == 0...10001
- **malloc(13):** 24 bytes, block header 0x00000019 == 0...11001

Block Placing and Splitting of Free Blocks

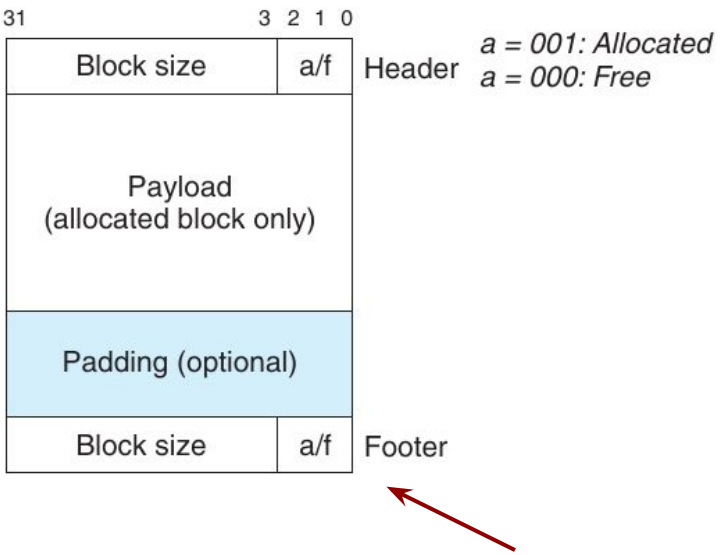
- If multiple blocks are available in the list, which one to pick?
  - **First Fit:** First block with enough space.
    ⇒ retains large blocks at the end of the list, but must skip many
  - **Next Fit:** First block with enough space, start from last position.
    ⇒ no need to skip small blocks at start, but worse memory utilization
  - **Best Fit:** Smallest block with enough space.
    ⇒ generally better utilization, but slower (must check all blocks)
- If available space is larger than required, what to do?
  - Assign entire block ⇒ internal fragmentation (ok for good fit)
  - Split the block at 2-word boundary, add another header

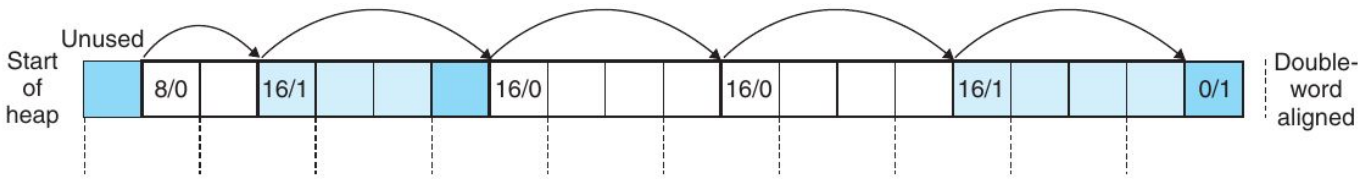


Getting additional memory

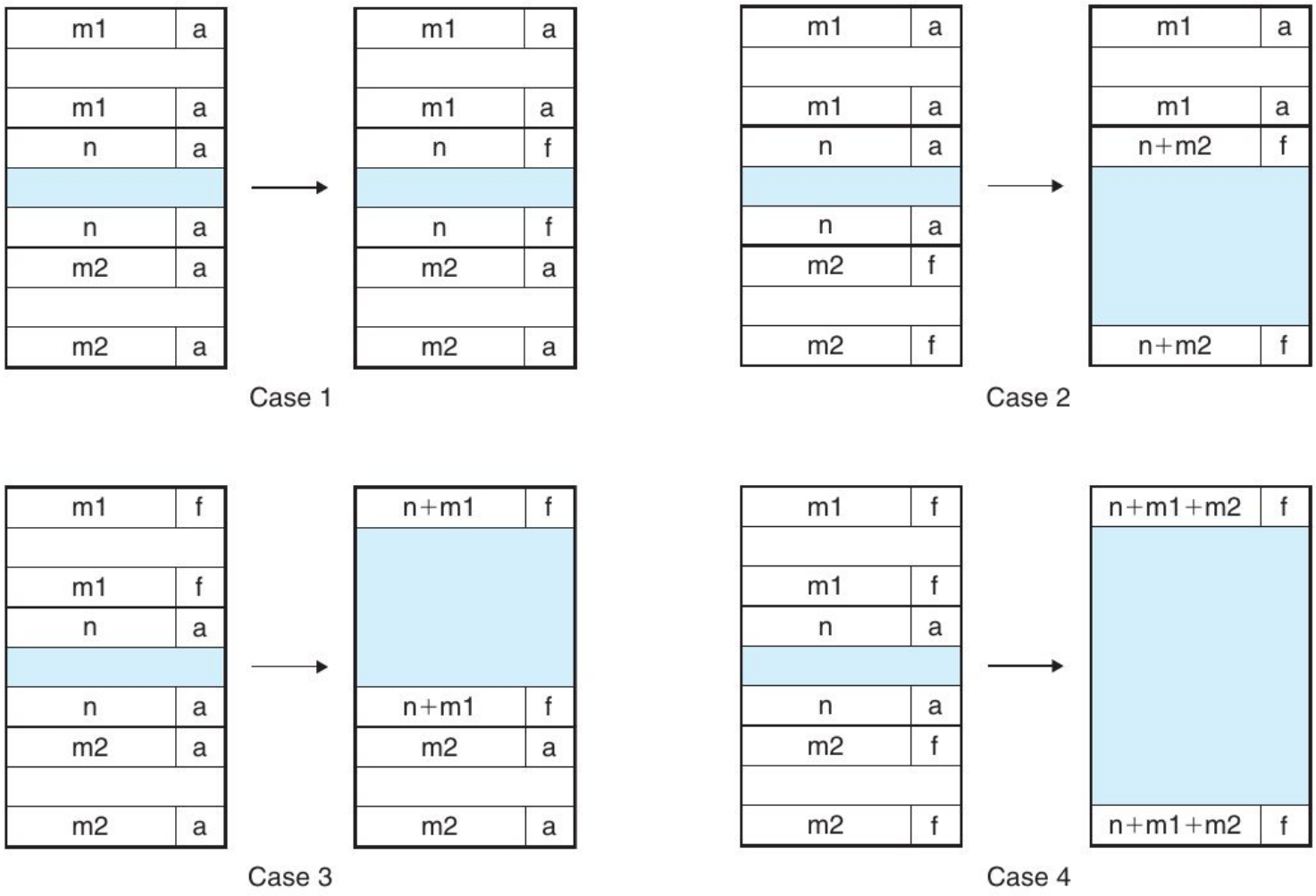
What to do when no free block is large enough?

- Extend the heap by calling `sbrk(intptr_t increment)`
  - Returns a pointer to the start of the new area
- Coalesce adjacent free blocks
  - Can coalesce both previous and following block
  - **Coalescing when freeing** blocks is O(1) but allows thrashing
  - **Boundary tag**
    Use a “footer” at the end of each (free) block to fetch previous block





Coalescing Cases



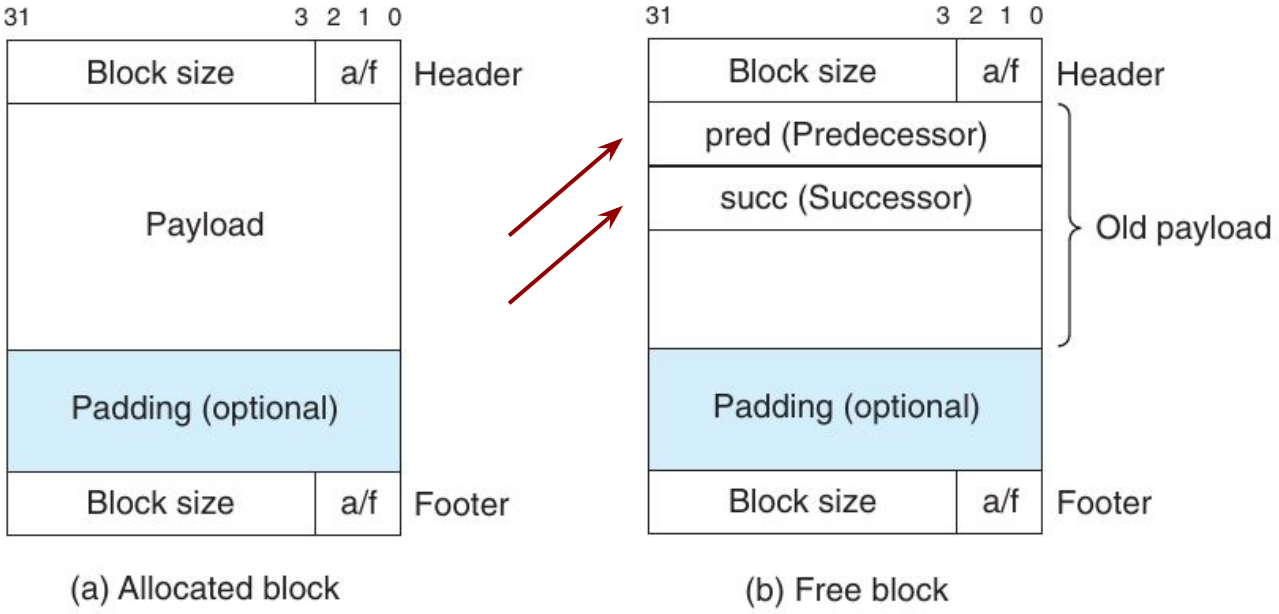
Explicit Free Lists

Allocation time is O(#blocks) for implicit lists...

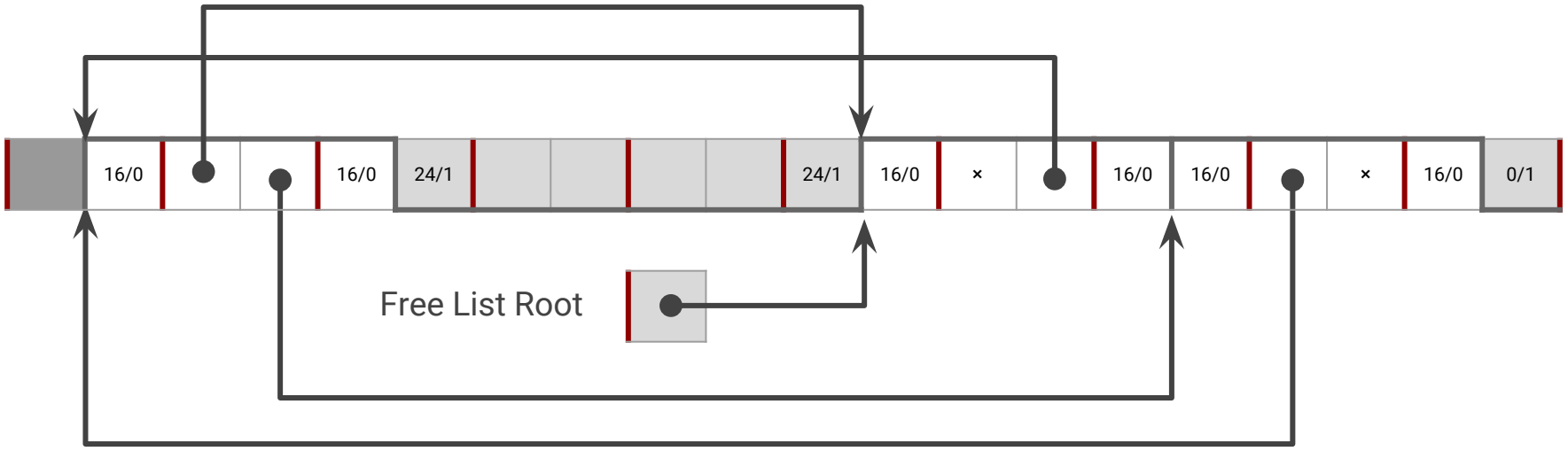
Idea. Organize free blocks as a doubly-linked list (**pointer inside free blocks**)

- LIFO ordering, first-fit placement ⇒ O(1) freeing/coalescing (boundary tags)
- Address order, first-fit placement ⇒ O(#free blocks) freeing

Much faster when memory is full, but lower memory utilization.



Explicit Free Lists: Example

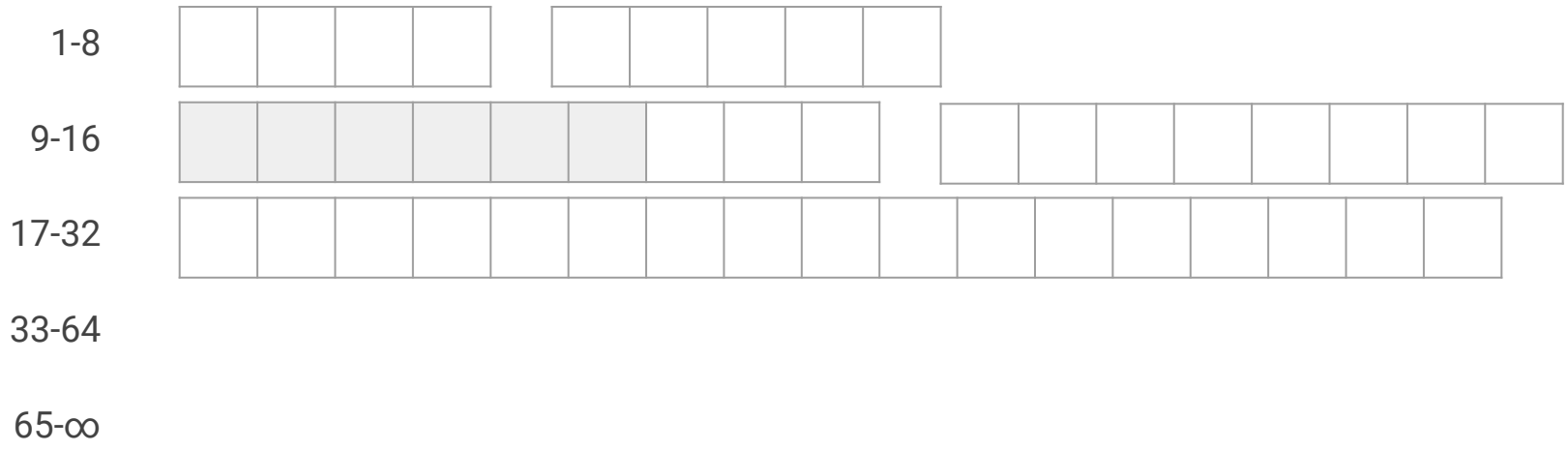


- Still need boundary tags for coalescing
- The next free block can be anywhere in the heap
- **Freeing with LIFO policy**
  - No coalescing: just add element to list head
  - Coalescing: remove contiguous blocks from free list, merge them, add the new free block to list head

Segregated Free Lists

To reduce allocation time:

- Partition free blocks into **size classes**, e.g., $(2^{(i)}, 2^{(i+1)}]$
- Keep a list of free blocks for each class
- Add freed blocks to the appropriate class
- Search a block of size *n* in the appropriate class, then following ones



Segregated Free Lists: Implementation

Simple Segregated

- Allocate maximum class size (e.g., $2^{(i+1)}$), never split free blocks
- Can assign first block from list and add freed blocks to front
- If list is empty, extend the heap and add blocks to list
- No header/footer required for allocated blocks, singly-linked free list

Segregated Fits

- First-fit search in appropriate class, split and add remaining to some class
- Extend appropriate class if also larger classes are empty
- To free a block, coalesce and place result in appropriate class

Advantages

- Higher throughput: log-time search for power-of-two size classes
- Higher utilization: first-fit with segregated lists is closer to best fit

Allocation Lab

Goal. Write your own version **`malloc`**, **`free`**, **`realloc`**.

- Understand their specification!
- Check the throughput / utilization effects of different strategies.

You only need to modify **`mm.c`**

```
#include <stdio.h>
extern  int  mm_init (void);          // init heap: 0 if successful, -1 if not
extern void *mm_malloc (size_t size);// 8-byte aligned non-overlapping heap region
extern void  mm_free (void *ptr);     // frees a pointer returned by mm_malloc
extern void *mm_realloc(void *ptr, size_t size);

/*
  mm_realloc(ptr, size)
    - if ptr is NULL, equivalent to mm_malloc(size)
    - if ptr is not NULL and size == 0, equivalent to mm_free(ptr)
    - if ptr is not NULL and size != 0, return pointer to area with new size
      (contract: old data unchanged, new data uninitialized)
*/
```

Support Routines

The **memlib.c** package simulates the OS memory system.

```
void *mem_sbrk(int incr);    // extend heap, return pointer to new area
void  mem_reset_brk(void);   // reset brk, release all heap memory
void *mem_heap_lo(void);     // address of first heap byte
void *mem_heap_hi(void);     // address of last heap byte == brk-1
size_t mem_heapsize(void);   // heap size in bytes
size_t mem_pagesize(void);   // system page size
void  mem_init(void);        // ignore (called by mdriver.c)
void  mem_deinit(void);      // ignore
```

Note that **mem_sbrk** accepts only a positive integer (cannot shrink the heap).

Recommended: Heap Checker

To debug, scan the heap

- Do allocated blocks overlap?
- Are blocks in the free list marked as free?
- Are all free blocks in the free list?
- Do pointers in the heap point to valid heap addresses
- Invent your own… but add comments!

Save the checks inside `int mm_check(void)` (return `0` if inconsistent)

- Style points will be given!
- During debug, call and quit if `mm_check() == 0`
- **Remove calls from final submission** (it would decrease throughput)

Evaluation: Trace Simulator

Trace Driver: `mdriver.c`

- Compile with `make`
- Checks correctness, space utilization, throughput
- `./mdriver -V` also prints debug information (for all traces)
- `./mdriver -f <trace>` to simulate a single trace

Trace File Format

- **Header**

```
<sugg_heapsize>   /* suggested heap size (unused) */
<num_ids>         /* number of request id's */
<num_ops>         /* number of requests (operations) */
<weight>          /* weight for this trace (unused) */
```

- **Requests**

```
a <id> <bytes>  /* ptr_<id> = malloc(<bytes>) */
r <id> <bytes>  /* realloc(ptr_<id>, <bytes>) */
f <id>          /* free(ptr_<id>) */
```

Trace Example

```
20000
2
5
1
a 0 512
a 1 128
r 0 640
f 1
f 0
```

Meaning

- Recommended heap size of 20,000 bytes (ignored)
- 2 distinct request IDs
- 5 requests
- Weight equal to 1 (ignored)
- Allocate 512 bytes, allocate 128, resize from 512 to 640, free all areas

Reusing code from CS:APP3e

Must add significant features to book implementation

- Explicit free list
- Segregated lists
- Deferred coalescing

You may use the macros, but remember to **cite your sources**.
To reiterate: changing the placement algorithm alone is not enough!

Additional rules

- You are not allowed to define global structs, arrays, lists, trees.
- Only global scalar values (integers, floats, pointers) are allowed.
- Returned pointers must be aligned to 8-byte boundaries.

Assigned Points

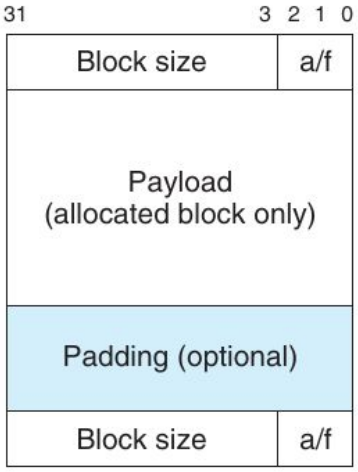
Breakdown

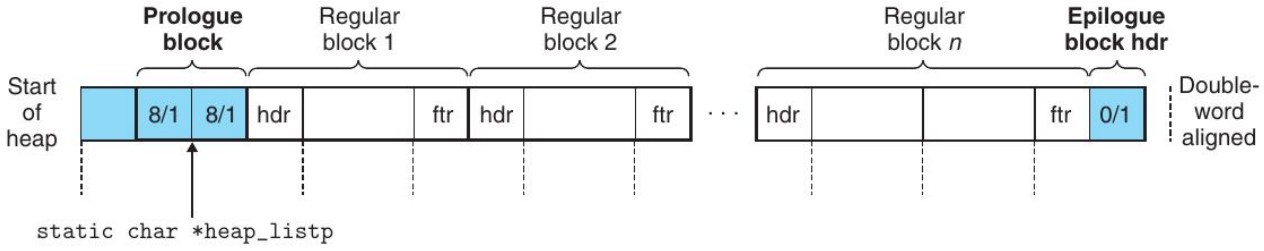
- 25 points for **correctness** (partial credit for each correct trace execution)
- 35 points for **performance**
  - **memory utilization** = peak memory usage / heap size (at most 1)
  - **throughput** = operations / second
  - **performance index** (*w* = 0.6)

$$P = wU + (1 - w)\min\left(1, \frac{T}{T_{libc}}\right)$$

Implementing an allocator (from CS:APP3e)

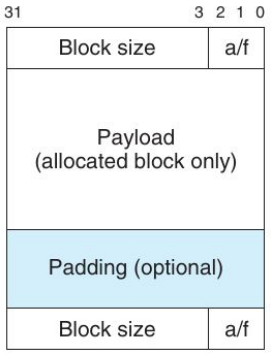
- Implicit free list with immediate boundary-tag coalescing
- 32-bit block size (4 GB), minimum block size of 16 byte
- Double-word aligned heap between **mem_heap** and **mem_brk**
- Call `mem_sbrk(intptr_t increment)` to increase **mem_brk**
- External API
  - `int mm_init(void) // 0 if successful, -1 otherwise`
  - `void *mm_malloc(size_t size)`
  - `void mm_free(void *ptr)`





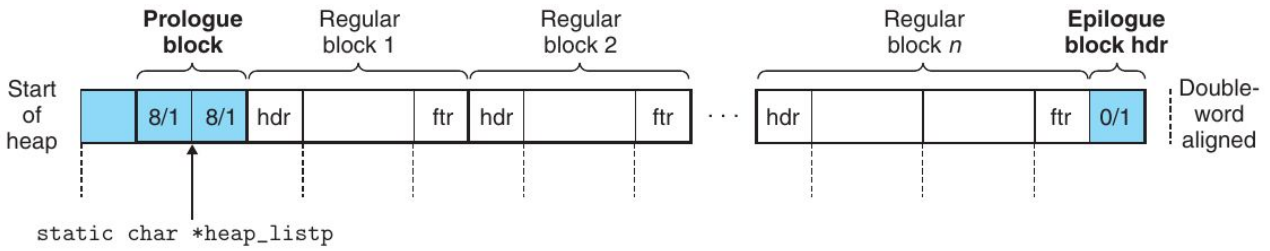
Constants and Macros

```
/* Basic constants and macros: notice the extra parentheses around arguments!  */
#define WSIZE 4                         /* Word and header/footer size (bytes) */
#define DSIZE 8                         /* Double word size (bytes)            */
#define CHUNKSIZE (1<<12)               /* Extend heap by this amount (bytes)  */
#define MAX(x, y) ((x) > (y)? (x) : (y))
/* Pack a size and allocated bit into a word (size has last 3 bits set to 0)   */
#define PACK(size, alloc) ((size) | (alloc))
/* Read and write a word at address p */
#define GET(p) (*(unsigned int *)(p))
#define PUT(p, val) (*(unsigned int *)(p) = (val))
/* Read the size and allocated fields from address p */
#define GET_SIZE(p) (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p) & 0x1)
/* Given block ptr bp, compute address of its header/footer */
#define HDRP(bp) ((char *)(bp) - WSIZE)
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)
/* Given block ptr bp, compute address of next and previous blocks */
#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
#define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))
```



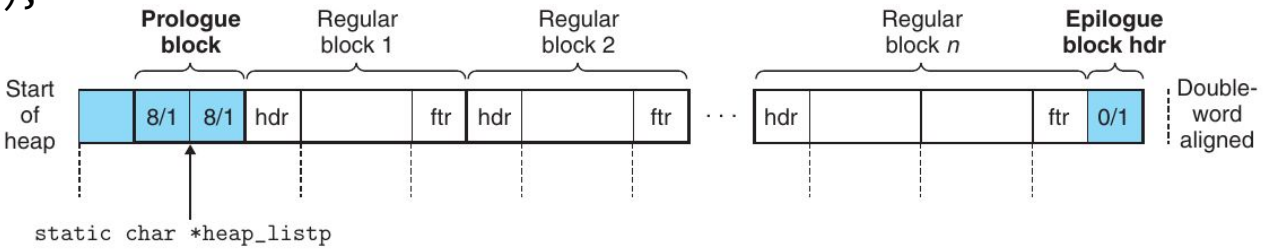
Initialize the heap

```
int mm_init(void) {
    /* Create the initial empty heap */
    if ((heap_listp = mem_sbrk(4*WSIZE)) == (void *)-1)
        return -1;

    PUT(heap_listp, 0);                          /* Initial 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;
}
```



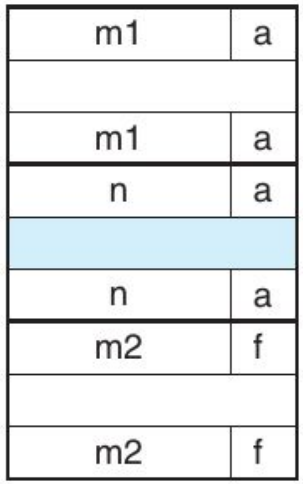
Extend the heap

```
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);
}
```



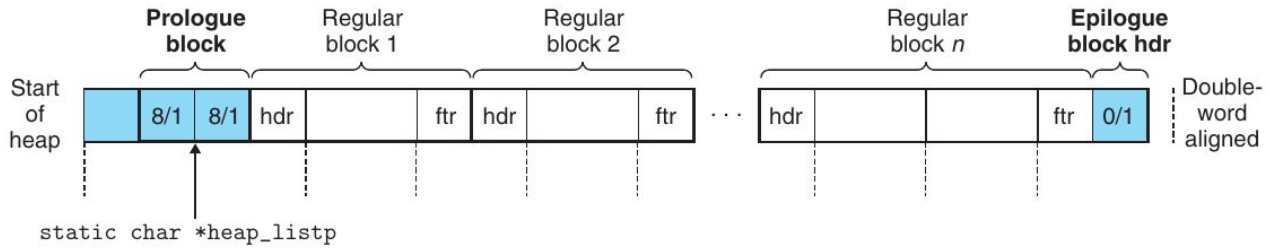
Coalesce

```
static void *coalesce(void *bp) {  /* Note: prologue/epilogue are marked as allocated */
    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;
    } 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;
}
```



Free

```
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);
}
```



Allocate

```
void *mm_malloc(size_t size) {
    size_t asize;      /* Adjusted block size */
    size_t extendsize; /* Amount to extend heap if no fit */
    char *bp;
    if (size == 0)     /* Ignore spurious requests */
        return NULL;

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

    if ((bp = find_fit(asize)) == NULL) {      /* Search free list for a fit */
        extendsize = MAX(asize, CHUNKSIZE);    /* Get more memory            */
        if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
            return NULL;
    }

    place(bp, asize);
    return bp;
}
```