CS356: Discussion #10

Dynamic Memory and Cache Lab

Illustrations from CS:APP3e textbook

USC University of Southern California

Cache Lab

Goal

- To write a small C simulator of caching strategies.
- Expect about 200-300 lines of code.
- Starting point in your repository.

Traces

- The `traces` directory contains program traces generated by `valgrind`
- The format of each line is: `<operation> <address>,<size>`
  For example: "`I 0400d7d4,8`" "`M 0421c7f0,4`" "`L 04f6b868,8`"
- Operations
  - Instruction load: `I` (ignore these)
  - Data load: `␣L` (hit, miss, miss/eviction)
  - Data store: `␣S` (hit, miss, miss/eviction)
  - Data modify: `␣M` (load+store: hit/hit, miss/hit, miss/eviction/hit)

http://bytes.usc.edu/cs356/assignments/cachelab.pdf

Reference Cache Simulator

```
./csim-ref [-hv] -S <S> -K <K> -B <B> -p <P> -t <tracefile>
```

`-h` Optional help flag that prints usage information
`-v` Optional verbose flag that displays trace information
`-S <S>` Number of sets (s=log2(S) is the number of bits used for the set index)
`-K <K>` Number of lines per set (associativity)
`-B <B>` Number of block size (i.e., use $B = 2^b$ bytes / block)
`-p <P>` Selects a policy, either LRU or FIFO
`-t <tracefile>` select a trace

```
$ ./csim-ref -S 16 -K 1 -B 16 –p LRU -t traces/yi.trace
hits:4 misses:5 evictions:3

$ ./csim-ref -S 16 -K 1 -B 16 –p LRU -v -t traces/yi.trace
 L 10,1 miss
 M 20,1 miss hit
 ... ...
 M 12,1 miss eviction hit
 hits:4 misses:5 evictions:3
```

(See https://usc-cs356.github.io/assignments/cachelab.html)

Reference Cache Simulator

LRU and FIFO

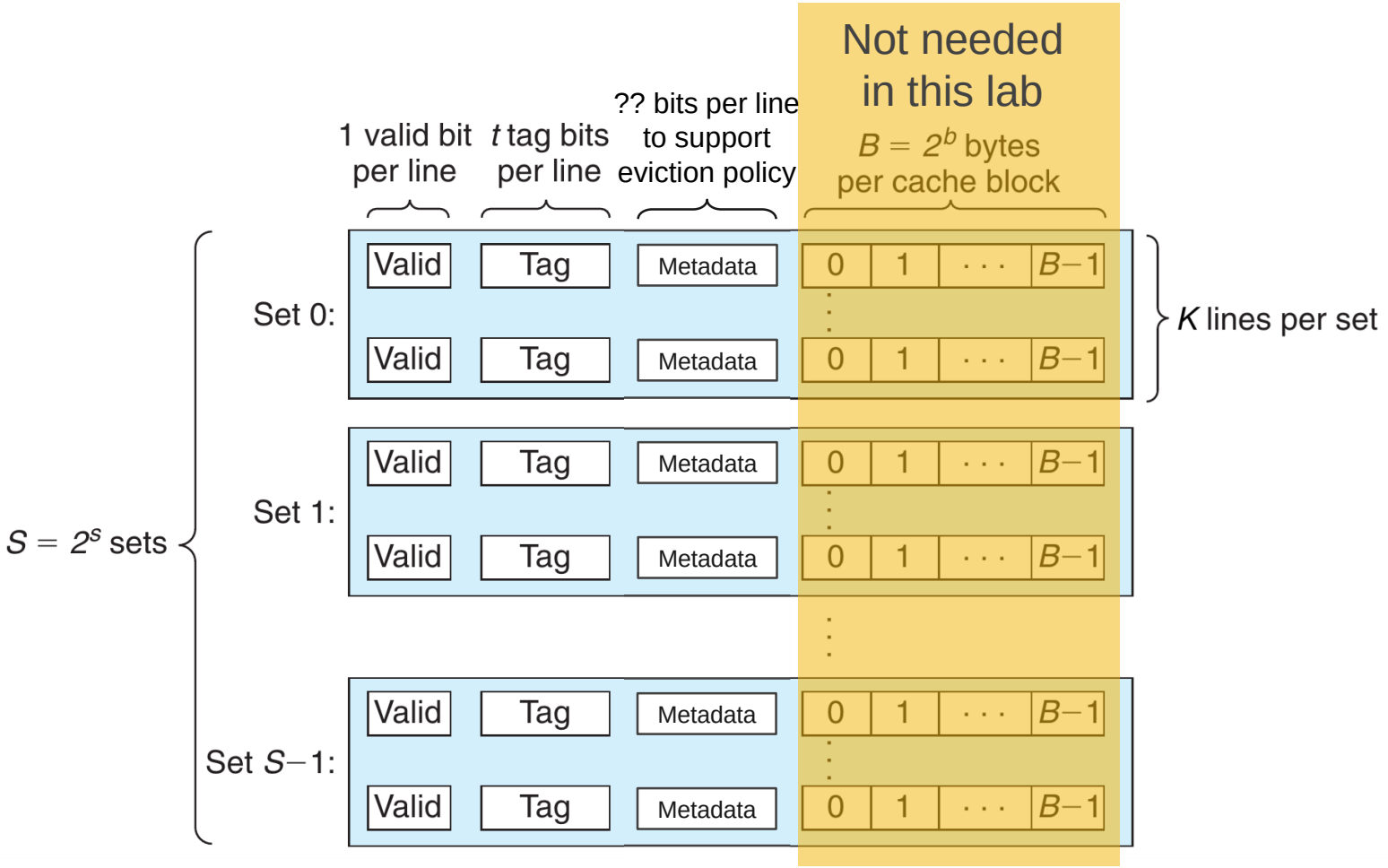
```
$ ./csim-ref -S 2 -K 2 -B 2 -p LRU -v -t traces/simple_policy.trace
 L 0,1 miss
 L 1,1 hit
 L 2,1 miss
 L 6,1 miss
 L 2,1 hit
 L a,1 miss eviction
 L 2,1 hit
hits:3 misses:4 evictions:1

$ ./csim-ref -S 2 -K 2 -B 2 -p FIFO -v -t traces/simple_policy.trace
 L 0,1 miss
 L 1,1 hit
 L 2,1 miss
 L 6,1 miss
 L 2,1 hit
 L a,1 miss eviction
 L 2,1 miss eviction
hits:2 misses:5 evictions:2
```

Reference Cache Simulator

Memory accesses that cross a line boundary

```
$ ./csim-ref -S 2 -K 1 -B 4 -p LRU -v -t traces/simple_size.trace
 L 0,2 miss
 L 1,2 hit
 L 3,1 hit
 L 6,6 miss miss eviction
 L 3,1 miss eviction
hits:2 misses:4 evictions:2
```

Your Simulator



Your Simulator

```
struct Line {
   // include valid bit, tag, and metadata
};
```

Example 1

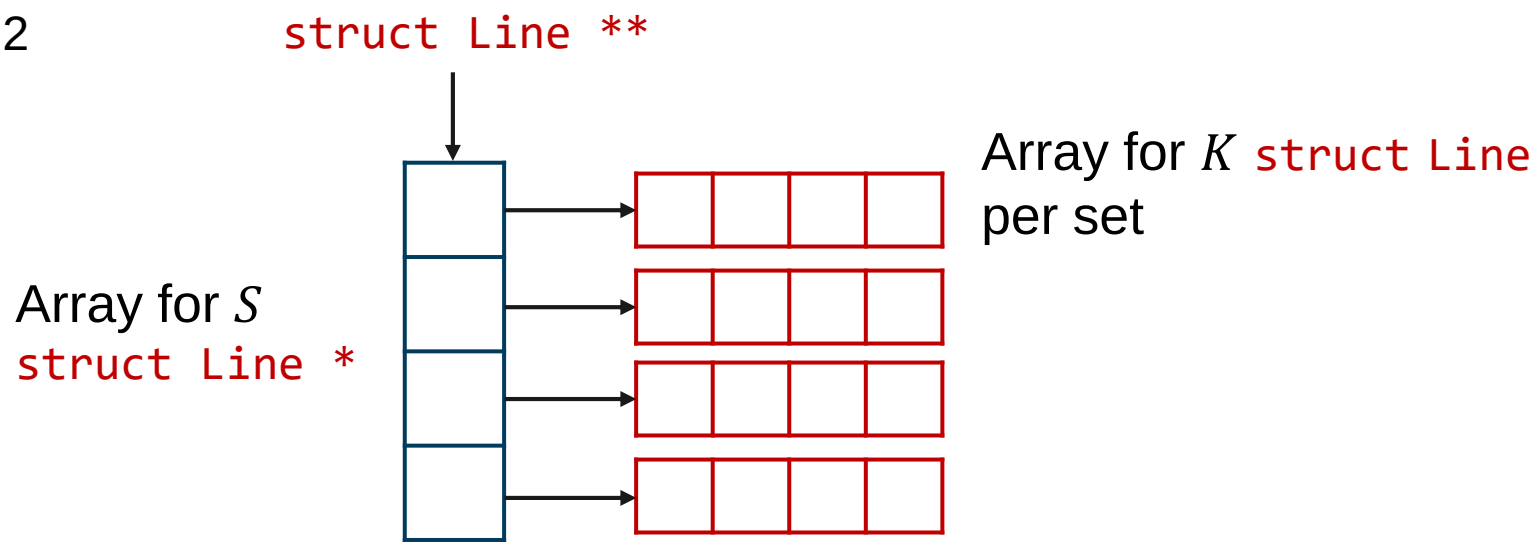
Flat array for $S \times K$ `struct Line`

```
struct Line *cache;
cache = (struct Line *)malloc(S * K * sizeof(struct Line));
```

Your Simulator

```
struct Line {
   // include valid bit, tag, and metadata
};
```

Example 2




```
struct Line **cache;
cache = (struct Line **)malloc(...);
for i = 0, 1, ..., S-1 do
    cache[i] = (struct Line *)malloc(...);
```

Your Simulator

Fill in the **`csim.c`** file to:

- Accept the same command-line options.
- Produce identical output.

Rules

- Include name and username in the header.
- Use only C code (must compile with **`gcc -std=c11`**)
- Use **`malloc`** to allocate data structures for arbitrary `S`, `K`, `B`
- Implement both LRU and FIFO policies.
- Ignore instruction cache accesses (starting with `I`).
- Memory accesses can cross block boundaries:
  ⇒ How to deal with this?
- At the end of your main function, call:

```
printSummary(hit_count, miss_count, eviction_count)
```

Evaluation

3 test suites:

- Direct Mapped: K = 1; no need to implement an eviction policy
- Policy Tests: check that LRU and FIFO policies work correctly
- Size Tests: include memory accesses that cross a line boundary

You only need to output the **correct number of cache hits, misses, evictions**.

- You can run **`csim-ref -v`** to check the expected behavior.
- Start from small traces such as `traces/dave.traces`
- Use the **`getopt`** library to parse command-line arguments.
  - `int s = atoi(arg_str); int S = pow(2, s);`

You must pass all tests in a test suite to receive its points.

Problems

- How to parse the input traces?
  - **`fopen`** (open a file), **`fgets`** (read a line), **`sscanf`** (parse a line), **`fclose`**
- How to represent the cache? How to allocate memory for any s, E, b?
  - Cache = **S sets**
  - Each set = **E cache lines**
  - Use **`malloc`** and **`free`**
- What needs to be stored in a cache line?
  - Valid bit, tag, and what else?
  - How to keep track of statistics for **LRU** and **FIFO** policies?
- How to retrieve data at a memory address?
  - How to extract tag / set / block bits from an input address?
  - How to select the correct set? And how to look for a hit?
  - What to update in case of hit (in addition to hit counter)?
  - What to do in case of miss?
- Useful: Print the content of the cache after each request in a trace

Dynamic Memory Allocation in C

Low-level memory allocation in Linux

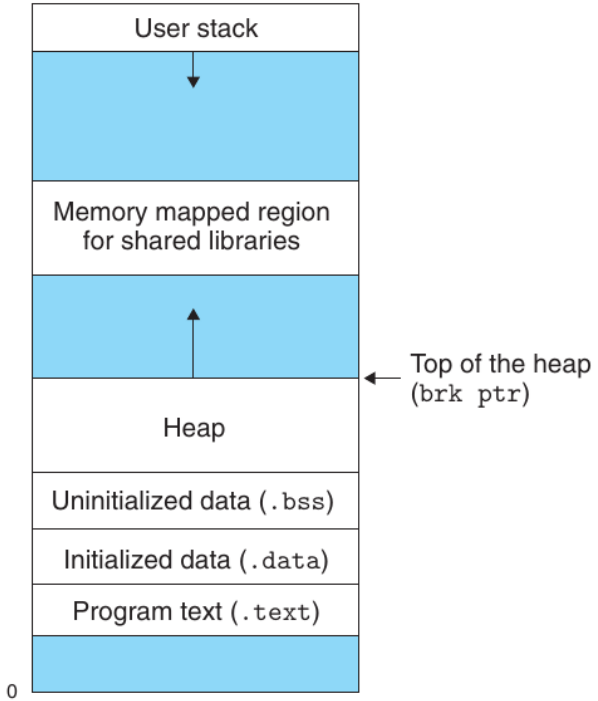
```
void *mmap(void *addr, size_t length,
  int prot, int flags, int fd, off_t off)
int munmap(void *addr, size_t length)
void *sbrk(intptr_t increment)
```

Portable alternatives in `stdlib`

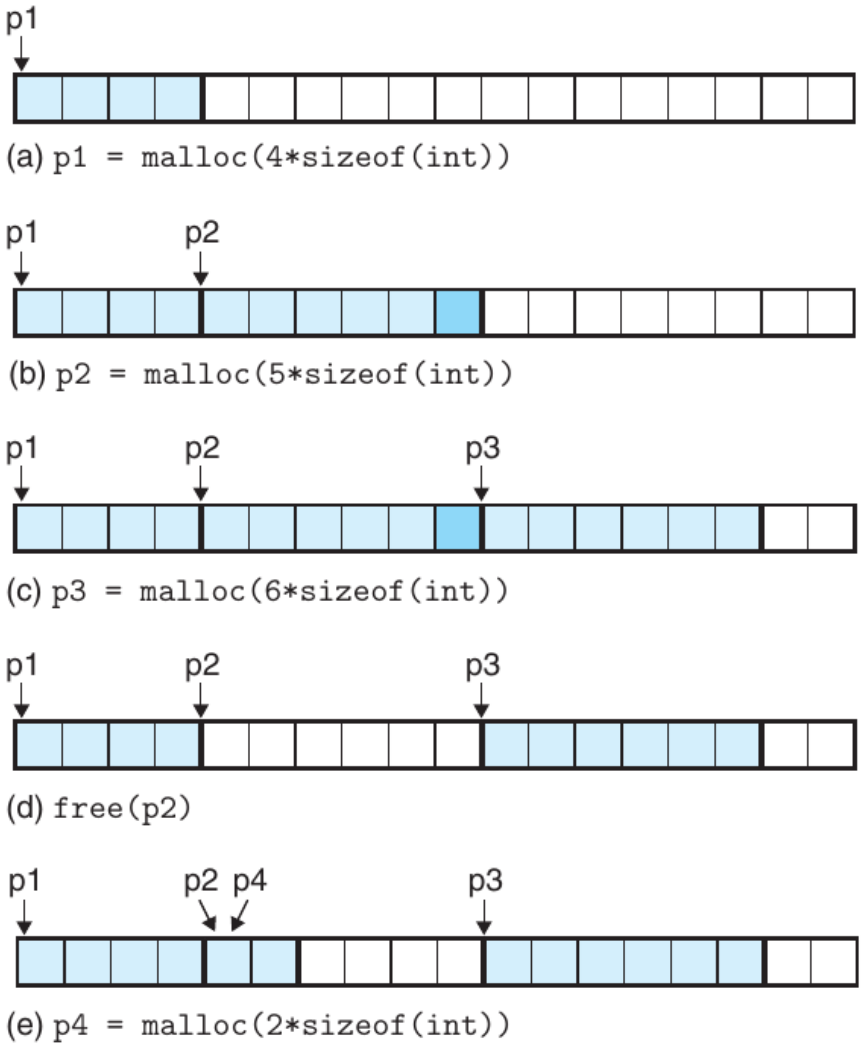
```
void *malloc(size_t size)
void *calloc(size_t nmemb, size_t size)
void *realloc(void *ptr, size_t size)
void free(void *ptr)
```

Check man pages for these functions!

```
$ man malloc
```



`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;
}
```

```
$ 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
```

```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;
}
```

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;
}
```

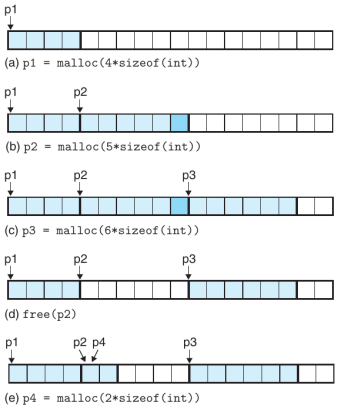
```c
int *binheapDelete(int **binheap, int *size) {
    int *packet = binheap[0];
    binheap[0] = binheap[*size - 1];
    *size--; /* This should be (*size)-- */
    heapify(binheap, *size, 0);
    return(packet);
}
```

```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**



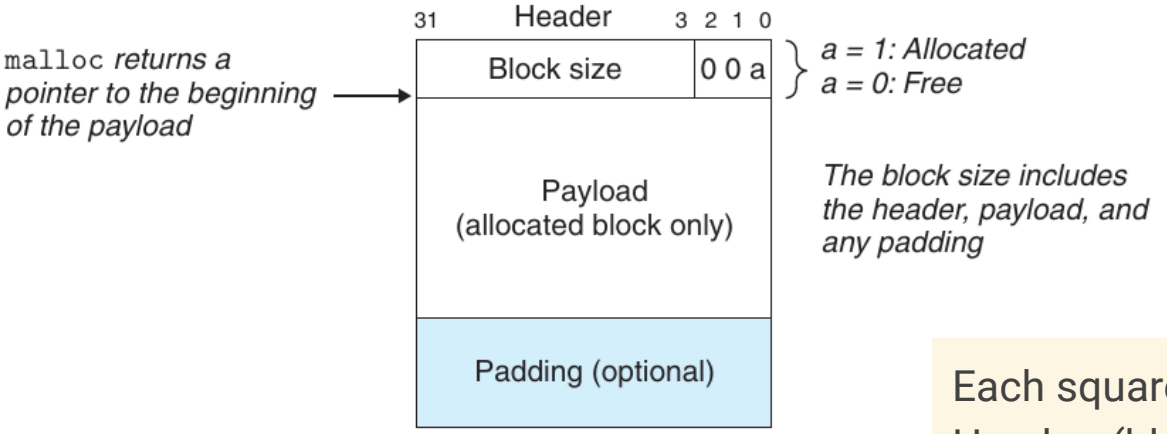
Goal 1. Maximize throughput: (# completed requests) / second

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

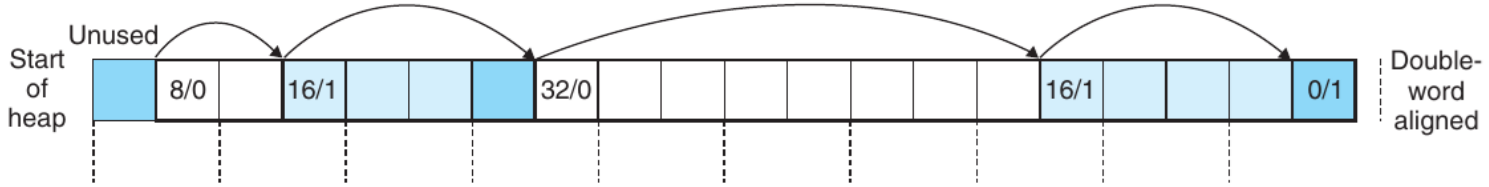
Goal 2. Maximize peak utilization: max{ allocated(t) : t $\leqslant$ T} / heapsize(T)

- If the heap can shrink, take max{ heapsize(t) : t $\leqslant$ 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)

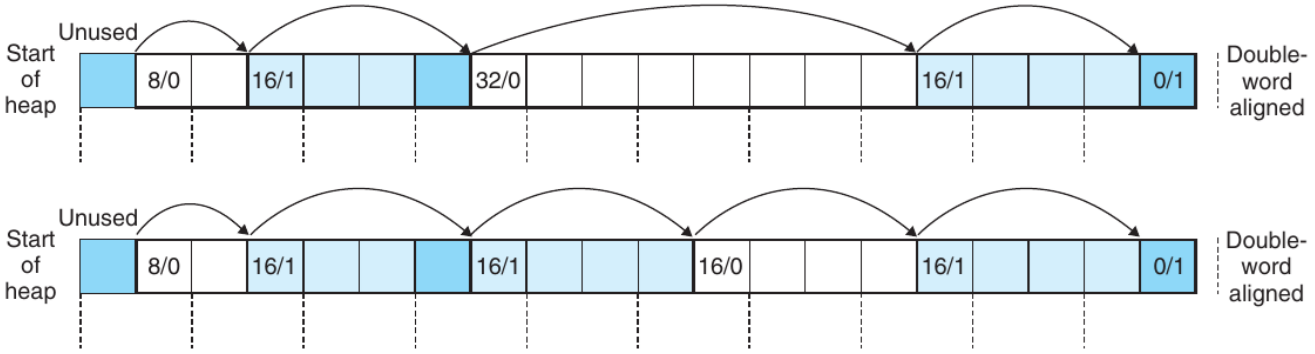
Block size (in bytes) and block header (in hex) for blocks allocated by the following sequence:

```
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

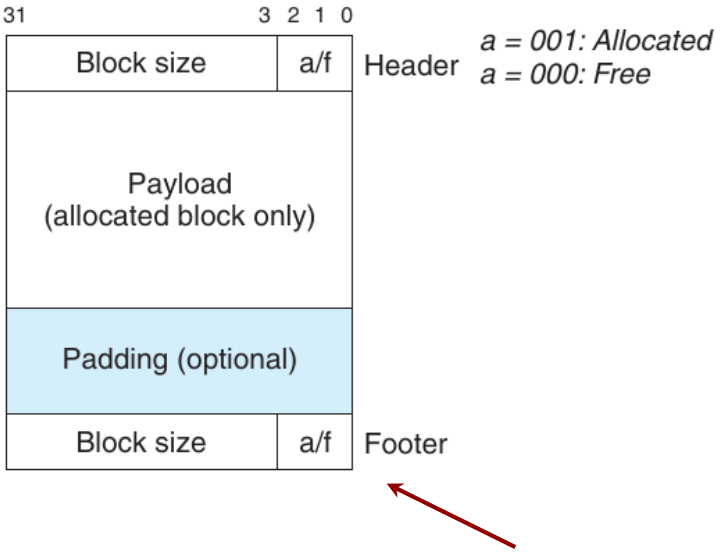
- If multiple blocks are available in the list, which one to pick?
  - **First Fit:** First block with enough space.
    ⇒ retains large blocks at the end of the list, but must skip many
  - **Next Fit:** First block with enough space, start from last position.
    ⇒ no need to skip small blocks at start, 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

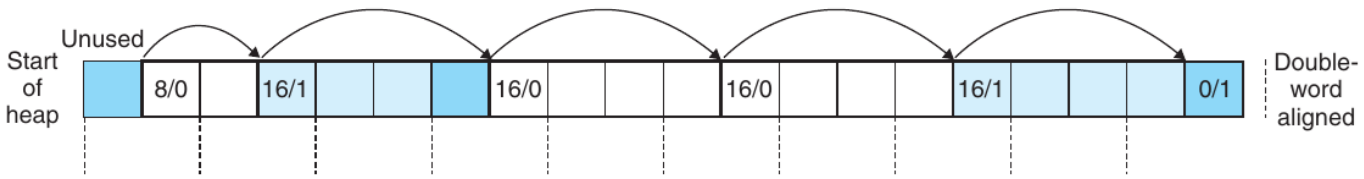


Getting additional memory

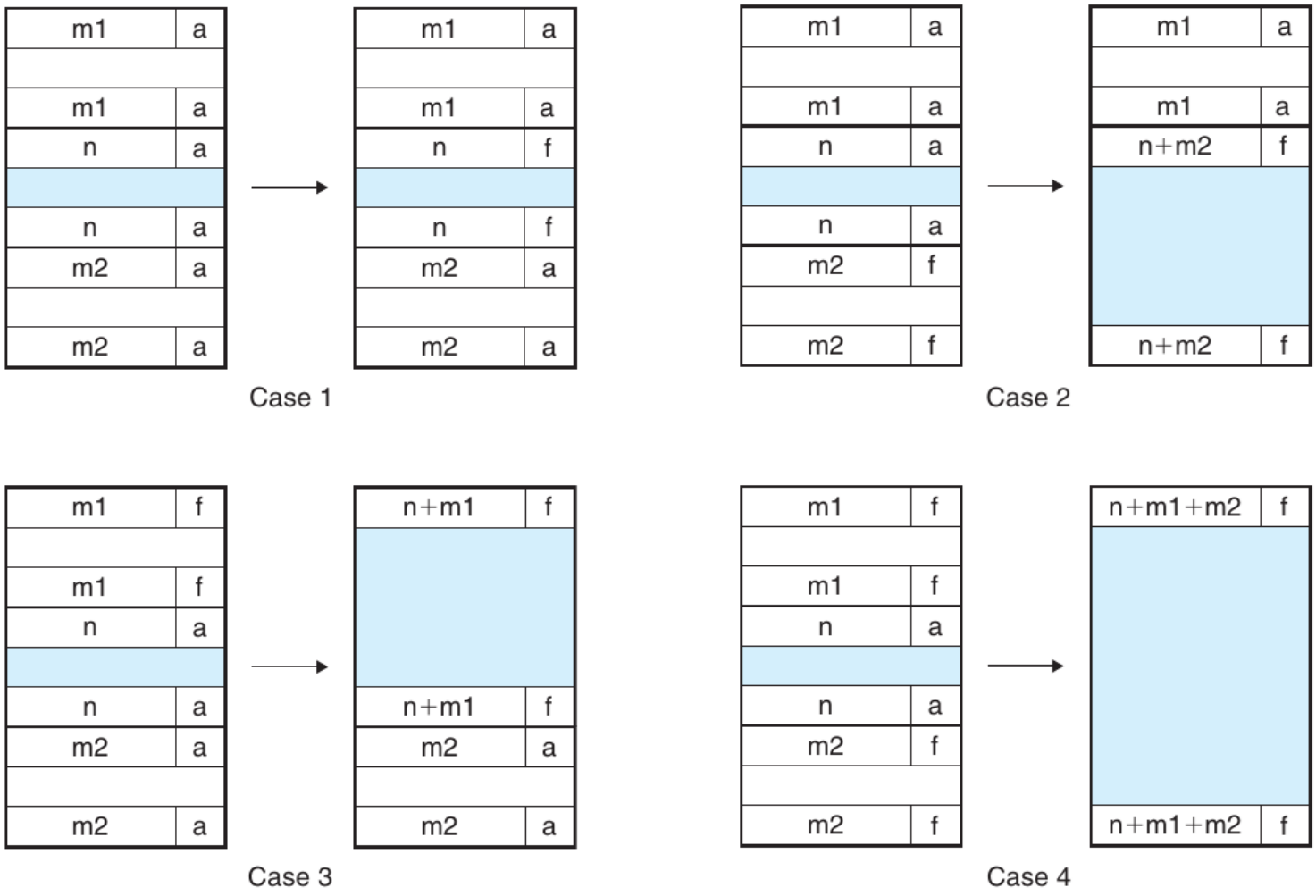
What to do when no free block is large enough?

- 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



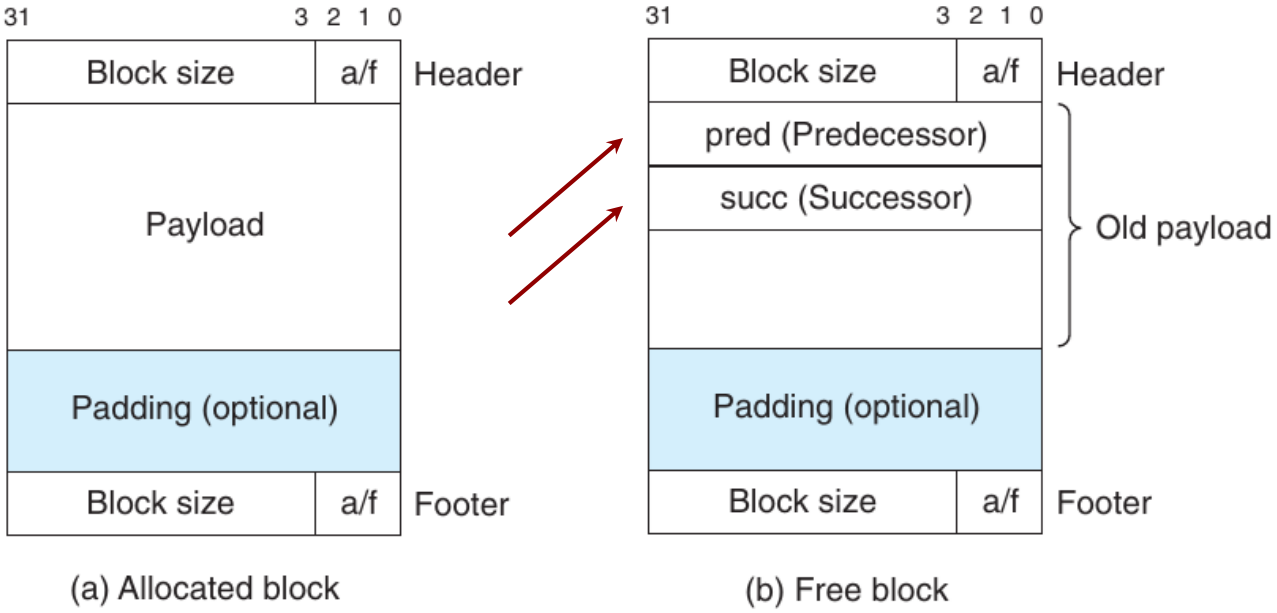
Explicit Free Lists

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

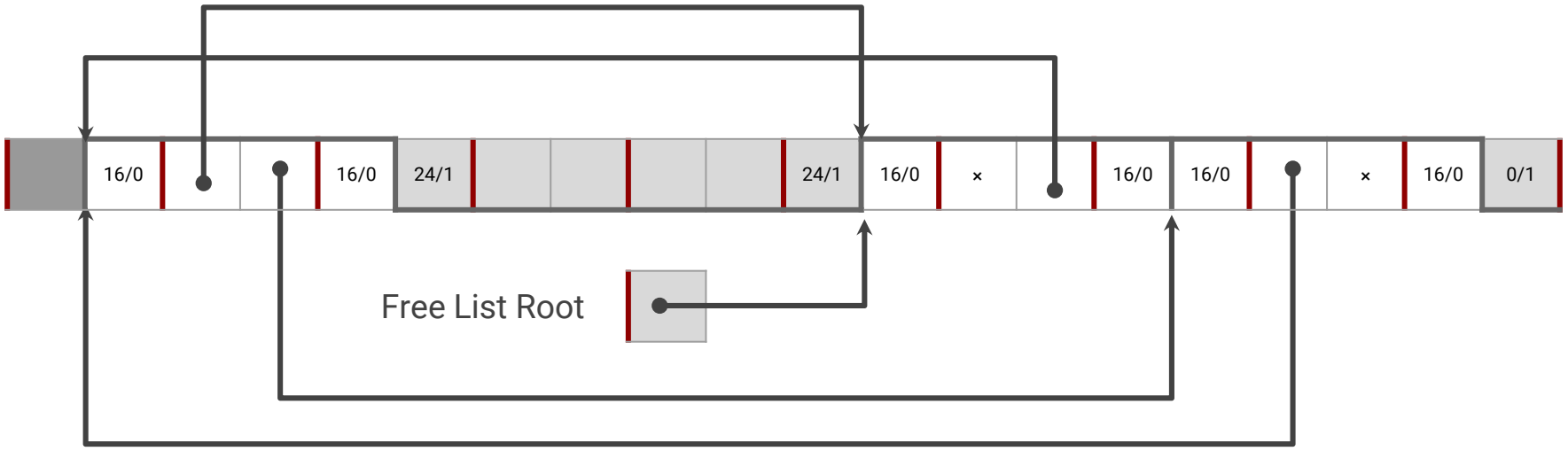
Idea. Organize free blocks as a doubly-linked list (**pointer inside free blocks**)

- 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

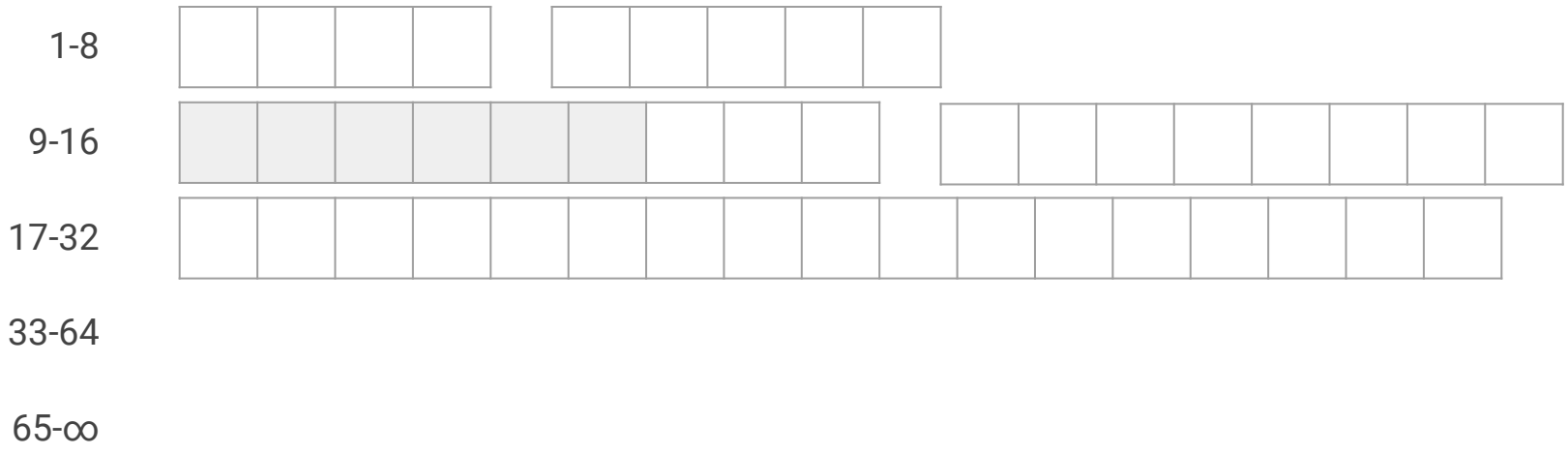


- Still need boundary tags for coalescing
- The next free block can be anywhere in the heap
- **Freeing with LIFO policy**
  - No coalescing: just add element to list head
  - Coalescing: remove contiguous blocks from free list, merge them, add the new free block to list head

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
- 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