CS 4400

Computer Systems

LECTURE 19

Dynamic memory allocation

Clicker Question

16-bit virtual addresses are translated to 13-bit physical addresses. The page size is 256 bytes. The TLB is 8-way set associative with 64 total entries. In which TLB set may the PTE (page table entry) for virtual address `0x6CA4` reside?

CLICK your one-digit answer.

Clicker Question

If a TLB miss occurs, the page corresponding to virtual address `0x6CA4` must be transferred from disk to main memory.

CLICK:

A. true

B. false

`mmap` Function

```
void* mmap(void *addr, size_t size,
           int prot, int flags,
           int fd, off_t offset);
```

- Allocates a new page; `size` should be a multiple of the page size.
- Suggest a virtual address in `addr`.
- Control other details via `prog` and `flags`.
- Use `fd` and `offset` to connect the page to a file.

`sbrk` Function (older, simpler than `mmap`)

```
void* sbrk(int incr);
```

- The *heap* is an area of memory that begins immediately after the `.bss` area and grows upward.
  - the kernel maintains variable `brk` as a pointer to the top
- The `sbrk` function grows or shrinks the heap by adding `incr` to the kernel's `brk` pointer.
- If successful, the old value of `brk` is returned.
- Else, -1 is returned and `errno` is set to `ENOMEM`.
- To get the current value of `brk`, call with `incr` = 0.

Dynamic Memory Allocator

- The `sbrk` and `mmap` functions are too primitive for most purposes.
- A *dynamic memory allocator* maintains the heap as a collection of various sized blocks.
  - each block is a contiguous piece of virtual memory
- Each block is designated as either *allocated* or *free*.
  - allocated—explicitly reserved for use by the application and remains so until explicitly freed (either by app or allocator)
  - free—available to be allocated and remains so until explicitly allocated by the application

Two Types of Allocators

- Both require the application to explicitly allocate blocks.
- *Explicit allocators*—require the application to explicitly free any allocated blocks.
  - *example*: `malloc` package in C
- *Implicit allocators*—require the allocator to detect when an allocated block is no longer being used by the application and then free the block.
  - AKA: garbage collectors (Java, C#, Racket, ...)
- For Lab 6, your task is to construct an explicit allocator.

`malloc` Function

```
void* malloc(size_t size);
```

- Returns a pointer to a block of memory of at least `size` bytes, suitably aligned for any kind of data object.
  - typically, `size_t` is `unsigned int` and 8-byte alignment
- If `malloc` encounters a problem, it returns NULL and sets `errno` appropriately.
  - e.g., requested block is larger than the available virtual memory
- To swap a previously allocated block with a block that is a different size, an application can use `realloc`.

```
void* realloc(void* ptr, size_t size);
```

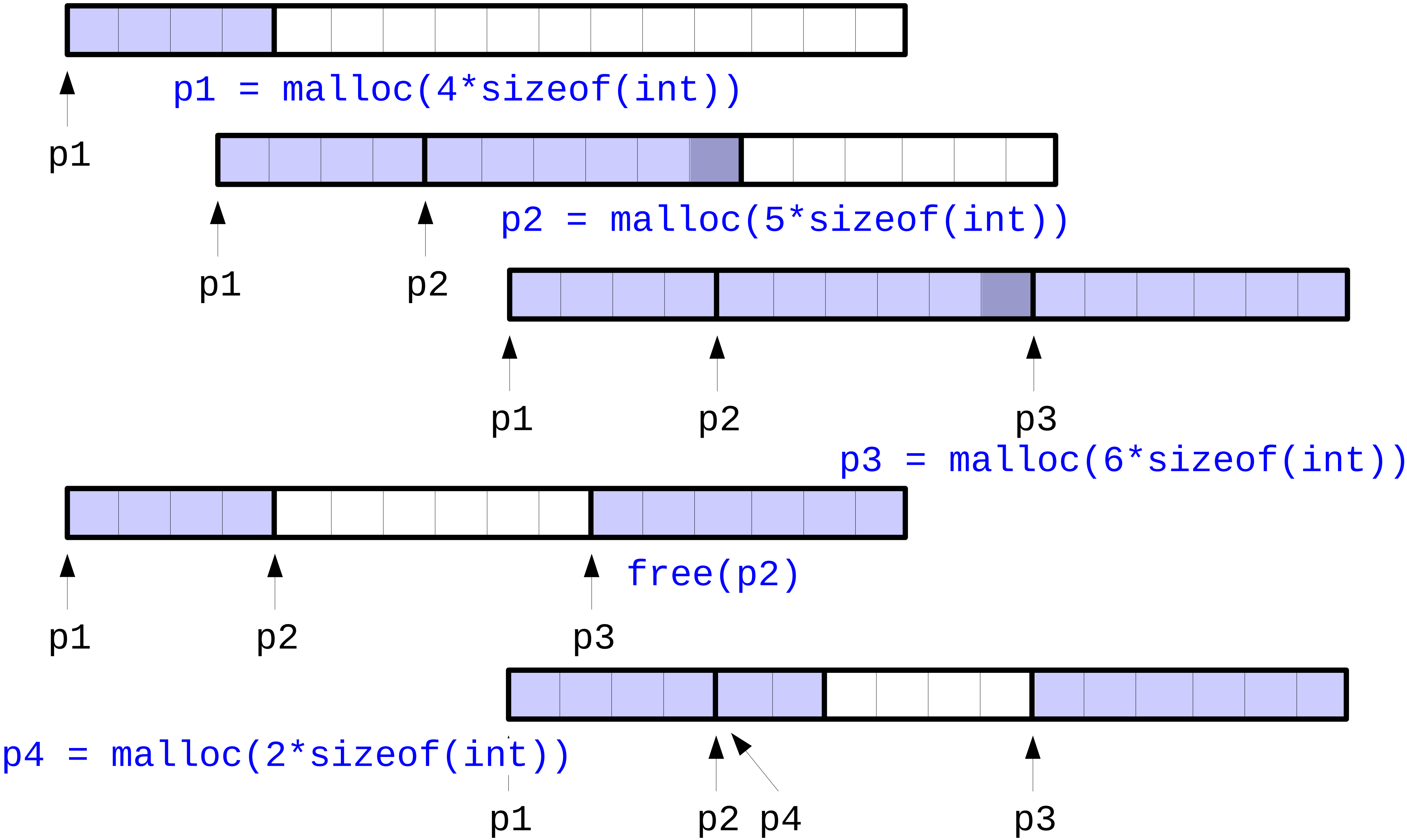
`free` Function

```
void free(void* ptr);
```

- Frees the allocated block indicated by `ptr`.
- If `ptr` does not point to the beginning of an allocated block (obtained from `malloc`), the behavior of `free` is undefined.
- Because `free` returns nothing, there is no indication to the application if something is wrong.

Example: `malloc`

16-word heap (initially one free block), each box is a 4-byte word, double-word alignment



Example: Dynamic Mem Alloc

```
#define MAXN 15213

int main() {
  int i, n;
  int array[MAXN];

  scanf(“%d”, &n);
  if(n > MAXN) {
    printf(“ERROR: too big\n”);
    exit(0);
  }
  for(i = 0; i < n; i++)
    scanf(“%d”, &array[i]);

  exit(0);
}
```

```
int main() {
  int i, n, *array;

  scanf(“%d”, &n);
  array = malloc(n*sizeof(int));
  for(i = 0; i < n; i++)
    scanf(“%d”, &array[i]);

  free(array);
  exit(0);
}
```

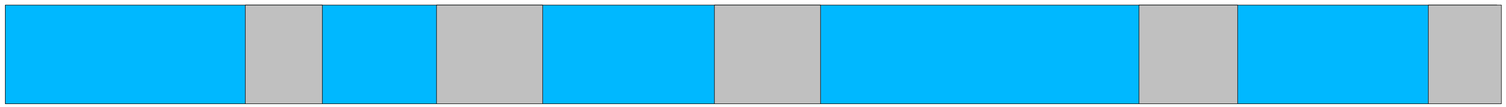
Explicit Allocator Requirements

- Cannot make any assumptions about the ordering of allocate and free requests.
  - cannot assume all allocate requests have matching free requests
- Must respond immediately to allocate requests.
  - cannot reorder or buffer requests to improve performance
- Must use the heap.
- Any allocated block must be aligned (typically 8-byte).
- Cannot modify or move blocks once they are allocated.

Explicit Allocator Goals

- Maximize throughput, i.e., the number of requests the allocator completes per unit of time.
  - 500 allocate and 500 free requests in 1 sec = 1000 ops per sec
  - minimize the average time to satisfy allocate and free requests
  - reasonable: linear-time allocate (worst case), constant-time free
- Maximize memory utilization.
  - virtual memory is limited
  - it is a finite resource that must be used efficiently
  - especially true if asked to allocate and free large blocks
- Finding the appropriate balance between these two goals is a challenge.

Fragmentation



- *Internal fragmentation*—occurs when an allocated block is larger than the payload.
  - because the allocator implementation imposes a minimum size
  - quantified as: sizes of allocated blocks – their payloads
- *External fragmentation*—occurs when there is enough free memory to satisfy an allocate request, but no single free block is large enough to handle the request.
  - depends on the pattern of previous request, as well as, the pattern of future requests (and allocator implementation)

Naïve Allocator Implementation

- Organize the heap as a large array of bytes and a pointer `p` that initially points to the first byte of the array.
- `malloc(size)`:
  - `old_p = p`
  - `p += size`
  - `return old_p`
- `free(ptr)`:
  - do nothing
- Throughput is extremely good. Why?
- Memory utilization is extremely bad. Why?

Implementation Issues

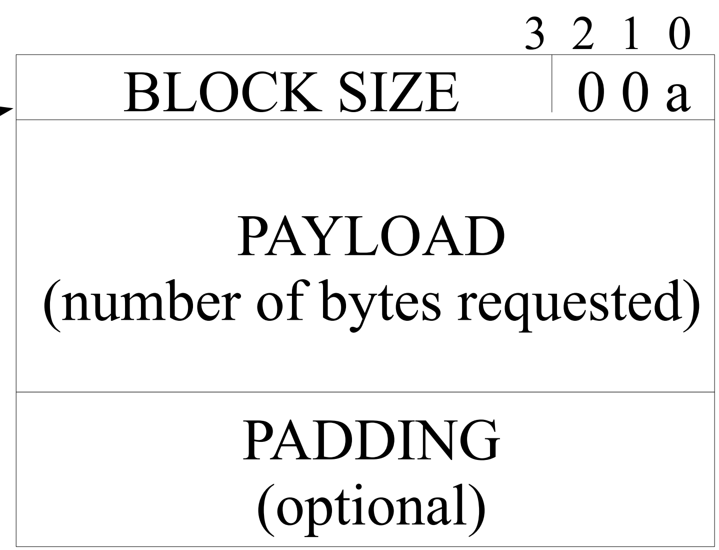
- *Free block organization*—how do you keep track of free blocks?
- *Placement*—how do you choose an appropriate free block in which to place a newly allocated block?
- *Splitting*—after placing newly allocated block in some free block, what do you do with the remainder of the free block?
- *Coalescing*—what do you do with a block that has just been freed?

Block Format

- An allocator needs a data structure for distinguishing between allocated and free blocks (and boundaries).

- This info may be embedded in the blocks themselves.

`malloc` *returns a pointer to the beginning of the payload*

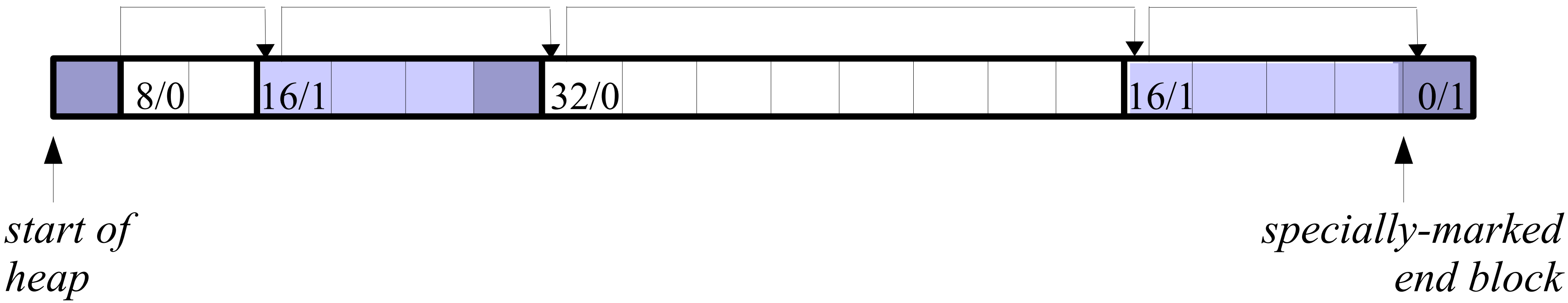


a = 1: *allocated*
a = 0: *free*

block size includes header, payload and any padding

- *Examples*:
  - an allocated block with size 24 bytes has header `0x00000019`
  - a free block with size 40 byes has header `0x00000028`

Implicit Free List



- Free blocks are linked implicitly by the size fields in the headers.

- The allocator can indirectly traverse the entire set of free blocks by traversing all of the blocks in the heap.

- *Pro*: simplicity, *Con*: cost of searching for a free block

Exercise: Block Format

- The minimum block size for an allocator is imposed by its alignment requirement and its block format.
- Determine the block sizes and header values the would result from the following `malloc` requests.
- *Assume*: double-word align, implicit free list, 4-byte headers
  - `malloc(1)`: 4 (header) + 1 (payload) + 3 (padding) = 8 bytes
    header = `0x8` | `0x1` = `0x9`
  - `malloc(5)`
  - `malloc(12)`
  - `malloc(13)`

CLICK the correct block header value:

A. `0x9`
B. `0xC`
C. `0xD`
D. `0x10`
E. `0x11`
F. `0x18`
G. `0x19`
H. `0x21`
I. none of above

Placing Allocated Blocks

- When a *k*-byte block is requested, the allocator searches the free list for a free block that is large enough.
  - the placement policy determines the manner of this search
- *First fit*—start at the beginning of the free list and choose first free block that fits.
- *Next fit*—start each search where previous search left off and choose the next free block that fits.
- *Best fit*—examine every free block and choose the smallest size that fits.

Placement Policies

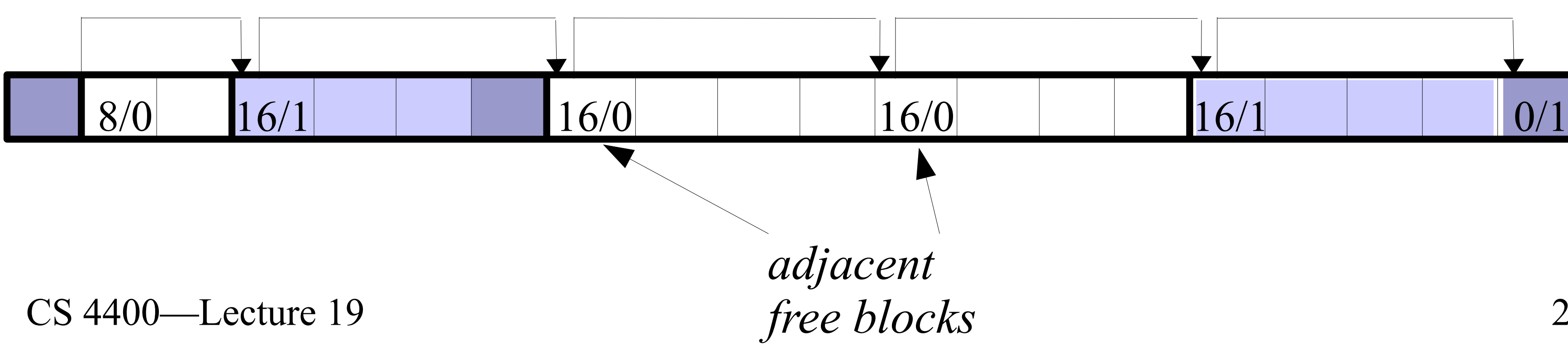
- First fit
  - *pro*: tends to retain large free blocks at the end of the list
  - *con*: tends to leave splinters of small free blocks at beginning of the list (increasing the search time for large blocks)
- Next fit
  - *idea*: if find fit in some block last time, good chance of finding fit in the remainder of the block next time
  - *pro*: can run significantly faster than first fit
  - *con*: studies suggest that memory utilization is worse
- Best fit
  - *pro*: studies show the memory utilization is the best
  - *con*: requires exhaustive search of the heap

Other Allocation Decisions

- Once a free block has been found that fits, how much of the free block should be allocated?
  - entire block—simple and fast, but introduces internal fragmentation
  - split the free block into two parts, allocated block and new free block
- What if the allocator is unable to find a fit?
  - create some larger free blocks by merging adjacent free blocks, if possible
  - ask the kernel for additional heap memory (`sbrk`), transform additional memory into one large free block in free list

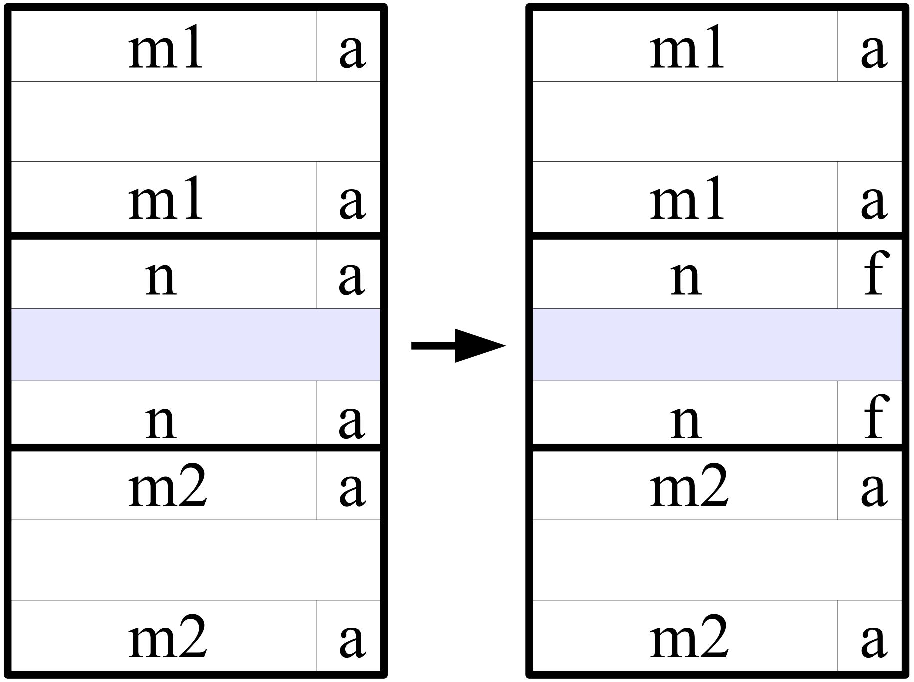
Coalescing Free Blocks

- When an allocated block is freed, there might be other free blocks that are adjacent to the newly freed block.
- *False fragmentation*—a lot of available free memory chopped up into small, unusable free blocks.
- *Coalescing*—merging adjacent free blocks.
  - immediate coalescing: performed each time a block is freed
  - deferred coalescing: waiting until some later time

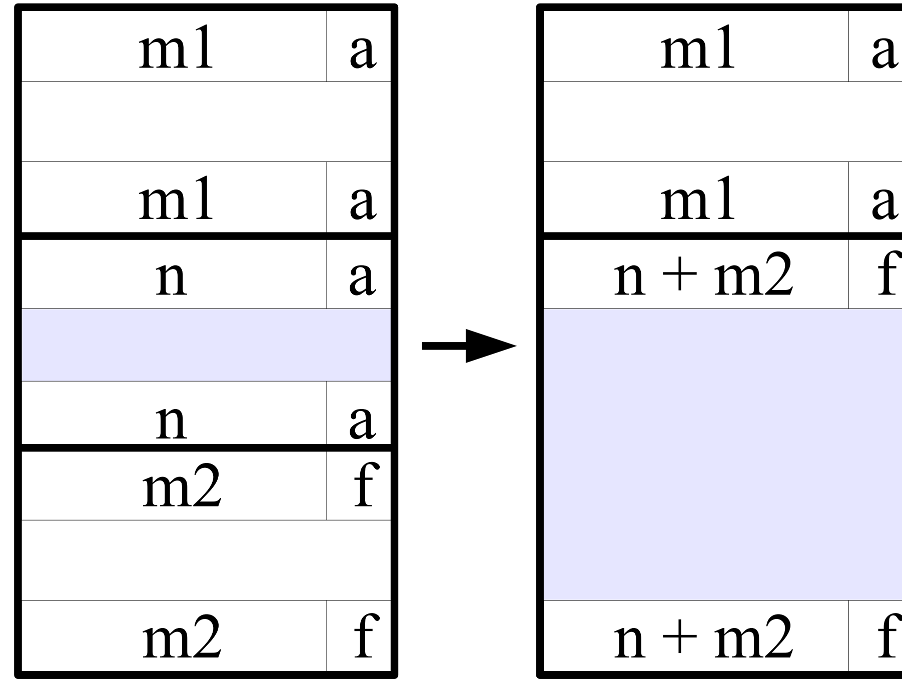


Boundary Tags

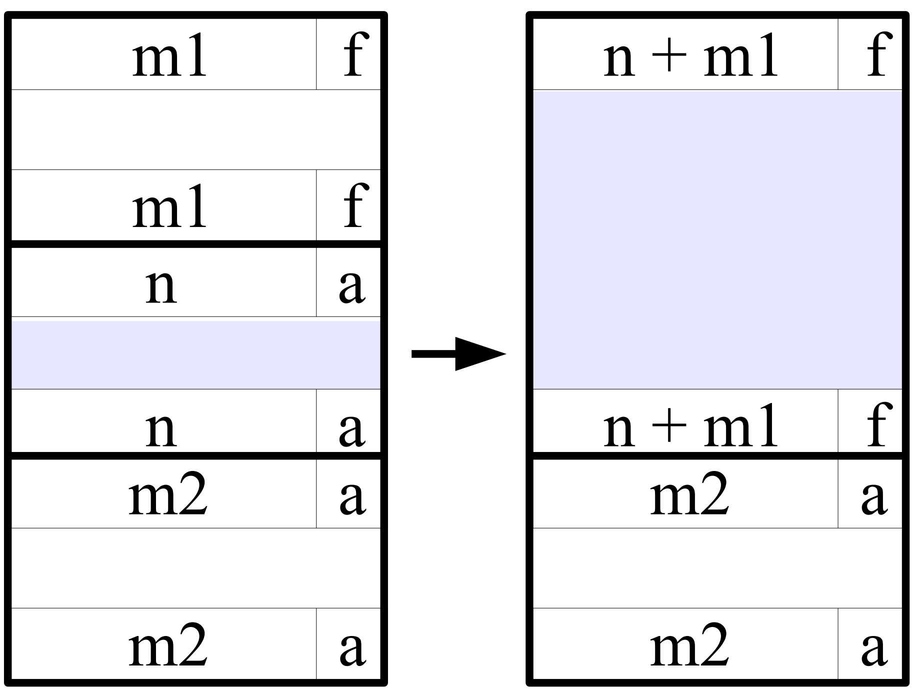
- Suppose we've just freed a block (the current block).
  - coalescing the next (free) block is straightforward
  - coalescing the previous (free) block requires a search
- Add a footer (the *boundary tag*) at the end of each block.
  - the footer is a replica of the header
- The allocator can determine the starting location and status of the previous block by looking at its footer.
  - only one word away from the start of the current block
- Is there a disadvantage to using boundary tags?
  - do allocated blocks really need footers?



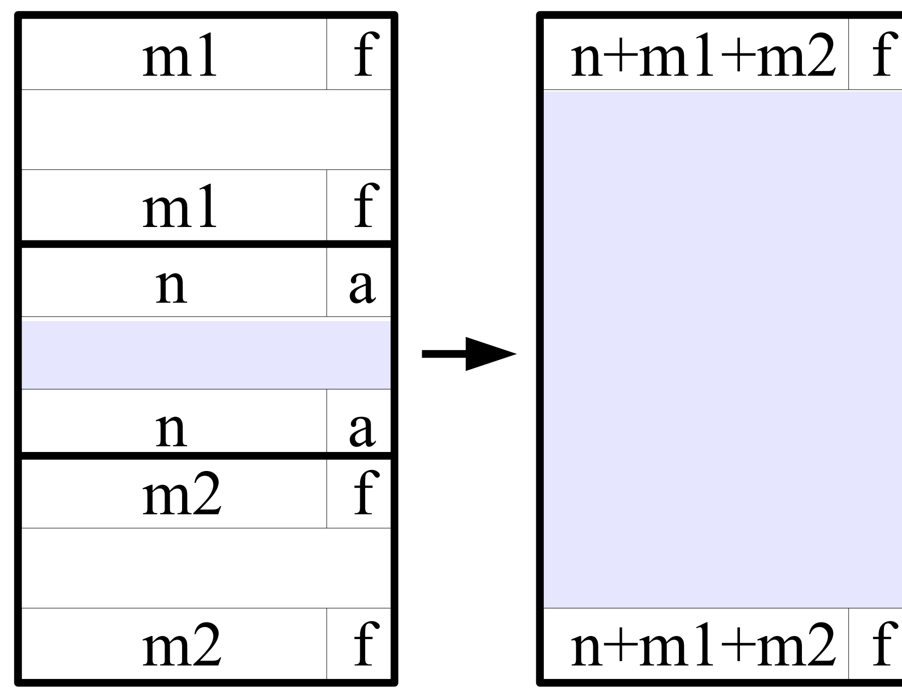
prev and next allocated



prev allocated, next free



prev free, next allocated



prev and, next free

Exercise: Minimum Block Size

Assume: implicit free list, headers/footers stored in 4-byte words, and every free block has a header and footer.

min block size = MAX(min allocated block size, min free block size)

- single-word alignment, allocated block has header and footer
  - alloc: 4-byte header, 1-byte payload, 4-byte footer – round up to 12
  - free: 4-byte header, 4-byte footer – 8
- single-word align, header only
- double-word align, header and footer
- double-word align, header only

CLICK the correct min block size:

A. 4 bytes
B. 8 bytes
C. 12 bytes
D. 16 bytes
E. none of the above