

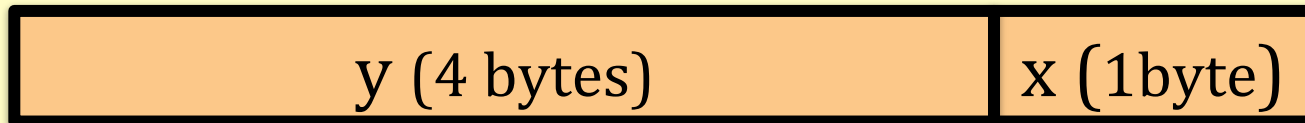
Lecture 6: Outline

1. Data alignment
2. Dynamic Memory Allocation in C
 - i. Sources of errors with DMA: memory leaks and dangling pointers
 - ii. Under the hood of malloc()
 - a) Performance goals of allocators
 - b) Implicit and explicit free lists
3. Linked lists

Representation in memory

```
struct p {  
    int y;  
    char x;  
};  
struct p sp;
```

sp



0x100

0x104

0x105

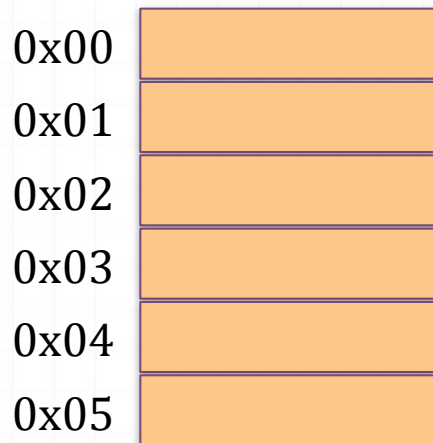
Data Alignment

```
struct p {  
    char x;  
    int y;  
};  
Struct p sp;
```

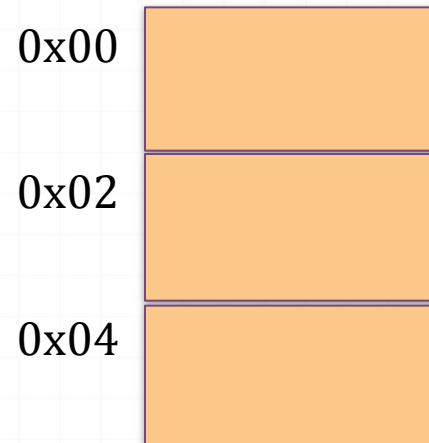
Q: How is the above struct laid out in memory?

Alignment Fundamentals

- Processors do not always access memory in byte sized chunks, instead in 2, 4, 8, even 16 or 32 byte chunks
- Boundaries at which data objects are stored affects the behavior of read/write operations into memory



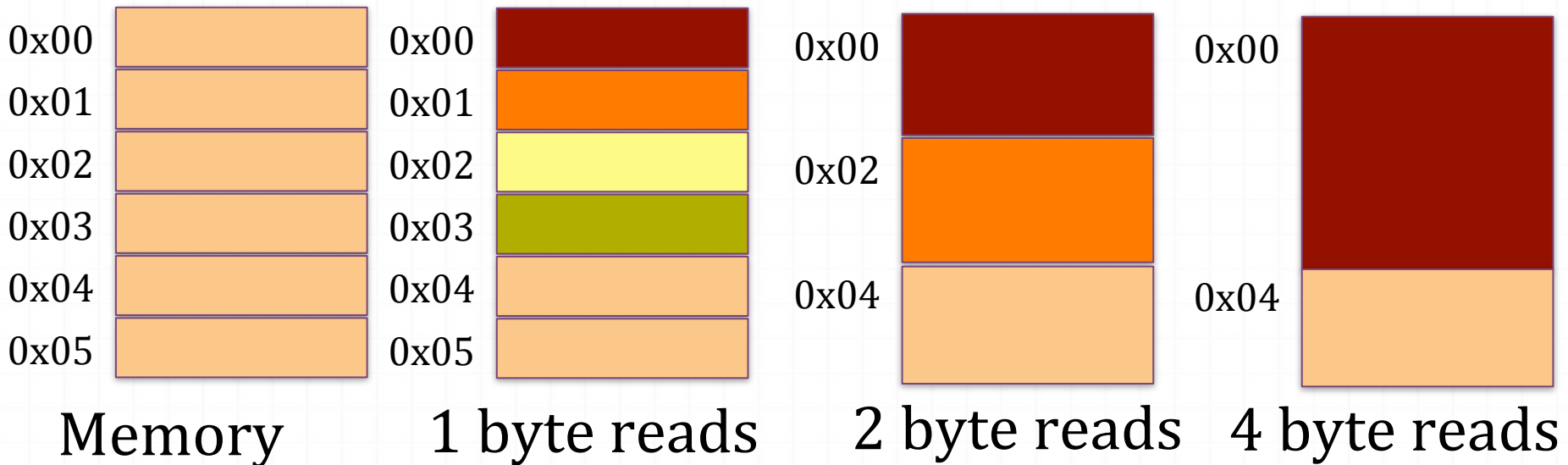
Programmer's view
of memory



Processor's view of
memory

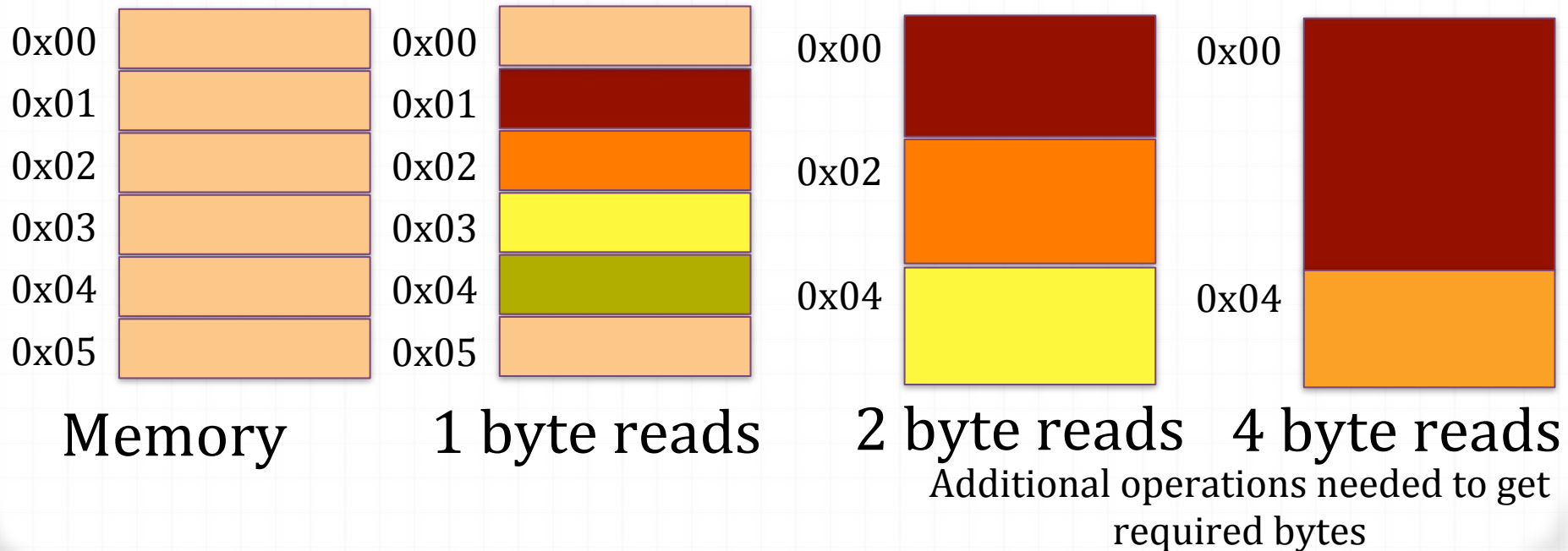
Alignment Fundamentals

- Consider the task of reading 4 bytes from memory starting at 0x00 for processors that read 1, 2, 4 bytes at a time.
- How many memory accesses are needed in each case?



Alignment Fundamentals

- Now consider the task of reading 4 bytes from memory starting at 0x01 for processors that read 1, 2, 4 bytes at a time.
- How many memory accesses are needed in each case?
 - Some processors just would not allow this scenario because it is extra work for the h/w and it affects the performance



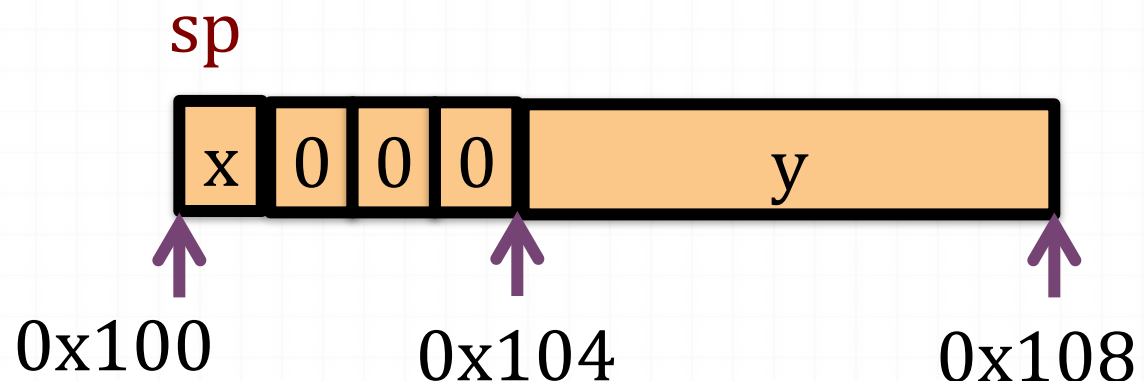
Alignment requirements

Data object of size X should start at address divisible by X.

- short (2 byte) at address divisible by 2 0b_____0
- int (4 byte) at address divisible by 4 0b_____00
- double (8 byte) at address divisible by 8 0b_____000

```
struct p {  
    char x;  
    int y;  
};  
struct p sp;
```

Q: How is 'sp' laid out in memory?
Compiler takes care of zero padding



How big are structs?

- Recall C operator `sizeof()` which gives size in bytes (of type or variable)

Q: How big is `sizeof(struct p)`?

```
struct p {  
    char x;  
    int y;  
};
```

- A. 5 bytes
- B. 8 bytes
- C. 12 bytes

Dynamic Memory Allocation (DMA)

1. Stack allocation:

- Dynamic, fast, easy but ...
- Follows last in first out discipline

2. Heap allocation: Lifetime of data determined by programmer

- More work on the s/w side, but
- Allows explicit control of lifetime of data

DMA on Heap

C library provides the following DMA functions:

A. Allocate memory on the heap:

`void * malloc(size_t size)`

Allocates a block of at least 'size' bytes and returns a pointer to the memory block

`void * calloc(size_t num, size_t size)`

Allocates a block of at least 'num*size' bytes & initializes the block to 0

`void * realloc(void *p, size_t s)`

Resizes the block to s,

Block may be moved to a different location on the heap

B. Free memory that was allocated before

`void free (void *p)`

DMA on Heap

- Key points:
 - Garbage collection not done for you in C
 - User is responsible to free memory that is not going to be used anymore
- Drawbacks: Large source of bugs in software
 1. Memory leaks: Memory in heap that can no longer be accessed
 2. Dangling reference : Pointer points to a memory location that cannot be dereferenced

This means that the location is either not a valid memory location or simply not available anymore

Memory Leak

- Memory leak: User allocates memory that is never freed
See example below:

```
void foo (int num)
{
    int *p = (int *) malloc(num *sizeof(int));
    return;
}
```

Programs memory usage keeps growing until its out of memory and crashes

Garbage collection: Frees memory that are no longer useful but we don't have a garbage collector in C

Which of the following is an example of a dangling pointer?

A.

```
void foo(int bytes) {  
    char *ch =(char *) malloc(bytes);  
    ....  
    free (ch);  
    ....}
```

B.

```
int * foo(int bytes) {  
    int i=14;  
    ....  
    return (&i);  
}
```

C.

```
char* foo(int bytes) {  
    char *ch =(char *) malloc(bytes);  
    ....  
    return (ch);  
}
```

D. A combination of the above

Which of the following is an example of a dangling pointer?

A.

```
void foo(int bytes) {  
    char *ch =(char *) malloc(bytes);  
    ....  
    free (ch);  
    ....}
```

B.

```
int * foo(int bytes) {  
    int i=14;  
    ....  
    return (&i);  
}
```

C.

```
char* foo(int bytes) {  
    char *ch =(char *) malloc(bytes);  
    ....  
    return (ch);  
}
```

D. A combination of the above

Q: Which of the following functions returns a dangling pointer?

```
int * f1(int num) {  
    int *mem1 =(int *)malloc(num*sizeof(int)) ;  
    return(mem1) ; }
```

```
int * f2(int num) {  
    int mem2[num] ;  
    return(mem2) ; }
```

- A.** f1
- B.** f2
- C.** Both

Q: Which of the following functions returns a dangling pointer?

```
int * f1(int num) {  
    int *mem1 =(int *)malloc(num*sizeof(int)) ;  
    return(mem1) ;}
```

```
int * f2(int num) {  
    int mem2[num] ;  
    return(mem2) ;}
```

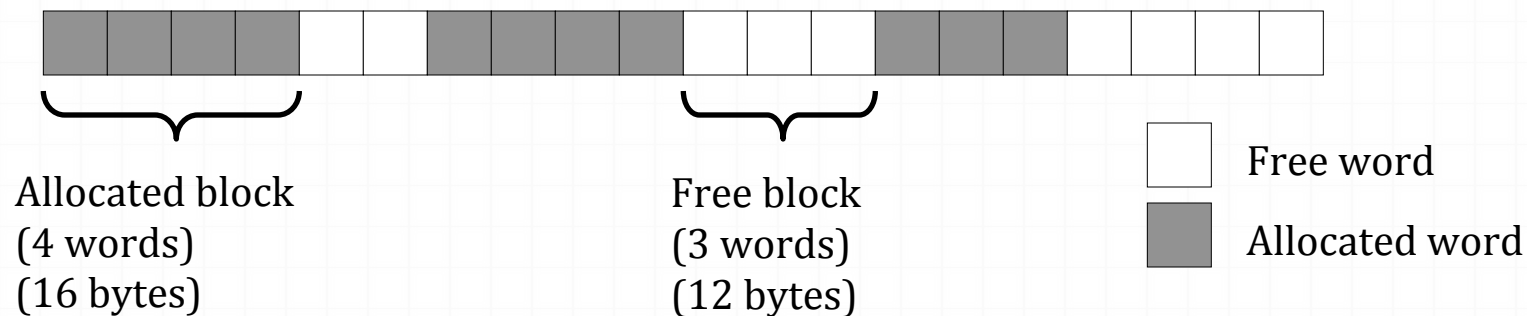
A. f1

B. f2 because mem2 is a local variable
created in the stack

C. Both

Under the hood of DMA in C

- The allocator has to keep track of free and used blocks
- Assumptions:
 - Memory is byte-addressed
 - Words are 4 bytes
 - Pointer fits in a word (4 bytes)
 - All diagrams are *word*-based



Allocation Examples

`p1 = malloc(16)`



`p2 = malloc(20)`



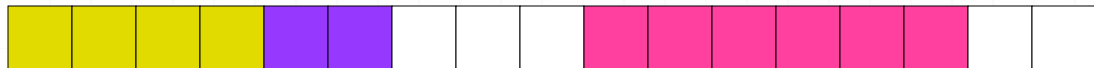
`p3 = malloc(24)`



`free(p2)`



`p4 = malloc(8)`



Which of the following is a valid behavior for a dynamic memory allocator? Why?

- A. Buffering memory allocation requests
- B. Re-ordering memory allocation requests
- C. Moving allocated memory to minimize wasted space
- D. Storing information in free blocks
- E. All of the above

Which of the following is a valid behavior for a dynamic memory allocator? Why?

- A. Buffering memory allocation requests
- B. Re-ordering memory allocation requests
- C. Moving allocated memory to minimize wasted space
- D. Storing information in free blocks
- E. All of the above

Constraints

- Applications:
 - Can issue arbitrary sequence of allocation and free requests
 - Must free blocks that were previously allocated
 - Cannot free memory that has already been freed
- Allocators
 - Must respond immediately to all allocation requests
 - i.e., can't reorder or buffer requests
 - Must allocate blocks from free memory
 - i.e., can only place blocks in memory that is currently free
 - Must align blocks so they satisfy all alignment requirements
 - 8-byte alignment for GNU malloc (`libc malloc`) on Linux boxes
 - Cant move the allocated blocks once they are allocated
 - i.e., compaction is not allowed

Performance Goals

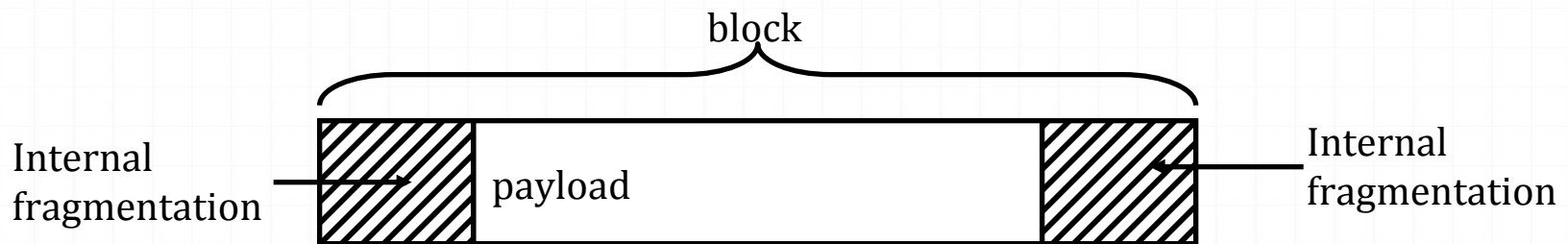
- Allocator tries to:
 - A. Maximize throughput
 - B. Minimize “wasted” space
 - These goals often conflict
- Throughput:
 - Number of completed requests per unit time
 - Example:
 - 5,000 malloc calls and 5,000 free calls in 10 seconds
 - Throughput is 1,000 operations/second.

Peak Memory Utilization (related to wasted space)

- Given some sequence of malloc and free requests:
 - $R_0, R_1, \dots, R_k, \dots, R_{n-1}$
- `malloc(p)` results in a block with a *payload* of p bytes
- *Aggregate payload P_k :*
 - After request R_k has completed, the *aggregate payload P_k* is the sum of currently allocated payloads—**excluding overhead**
- *Current heap size is denoted by H_k*
 - Assume that H_k is monotonically non-decreasing
- *Peak memory utilization:*
 - After k requests, *peak memory utilization* is:
 - $U_k = (\max_{i < k} P_i) / H_k$

Internal Fragmentation

- Poor memory utilization is caused by *fragmentation*.
 - Comes in two forms: internal and external fragmentation
- Internal fragmentation –
 - For any block, internal fragmentation is the difference between the block size and the payload size



- Caused by overhead of maintaining heap data structures, padding for alignment purposes, or explicit policy decisions (e.g., not to split a block)

External Fragmentation

Occurs when there is enough aggregate heap memory, but no single free block is large enough

```
p1 = malloc(16)
```



```
p2 = malloc(20)
```



```
p3 = malloc(24)
```



```
free(p2)
```



```
p4 = malloc(24)
```

oops!

Implementation Issues

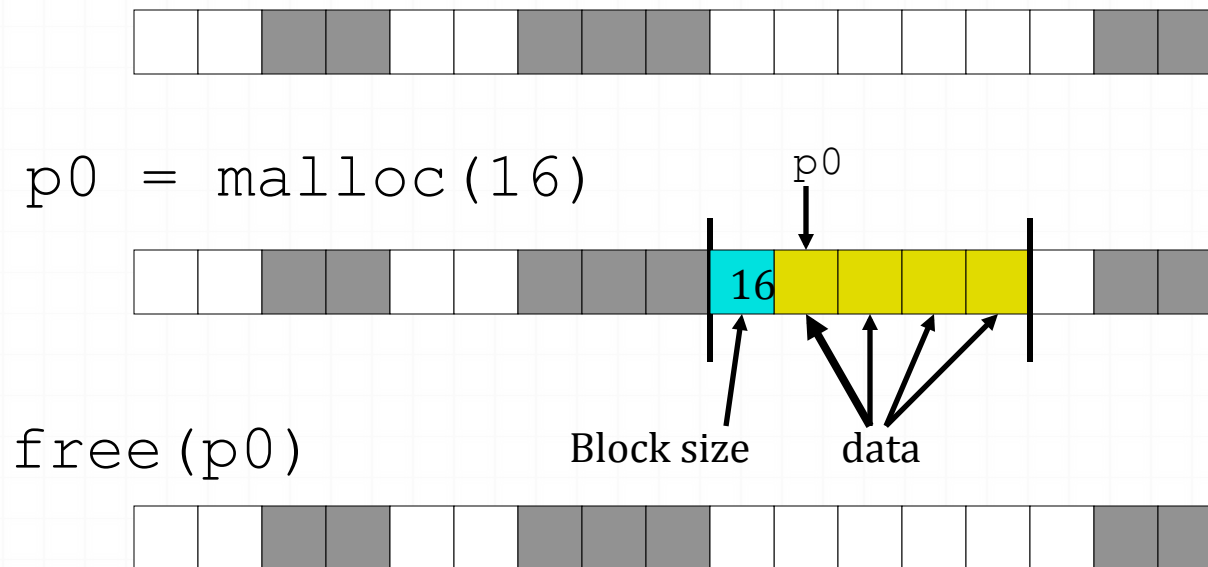
1. How do we know how much memory to free when given just a pointer?
2. How do we track free blocks? (“free list”)
3. What to do with extra space when allocating a structure smaller than the free block it is placed in?
4. How do we pick a block to use for allocation?
(Many might fit) : Policies : First fit, next fit, best fit
5. How do we reinsert freed block in free list?

Implementation Issues

1. How do we know how much memory to free when given just a pointer?
2. How do we track free blocks? (“free list”)
3. What to do with extra space when allocating a structure smaller than the free block it is placed in?
4. How do we pick a block to use for allocation?
(Many might fit) : Policies : First fit, next fit, best fit
5. How do we reinsert freed block in free list?

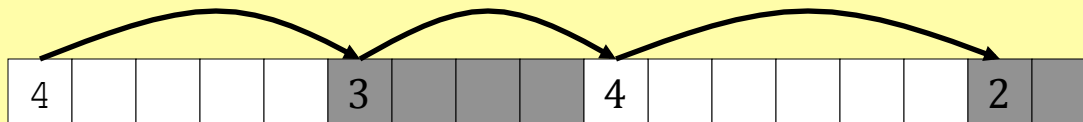
Knowing How Much to Free

- Standard method
 - Keep length of block in preceding word
 - Often called *header field* or *header*
 - Requires extra word for every allocated block



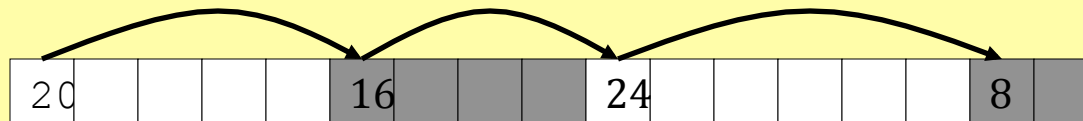
Keeping Track of Free Blocks

- Method 1: *Implicit list* using lengths -- links all blocks

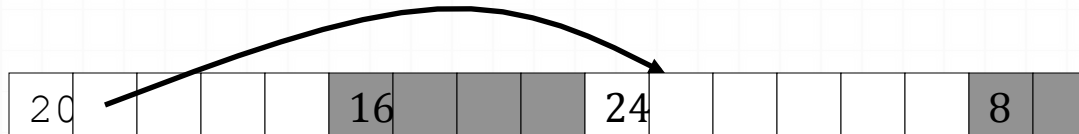


Keeping Track of Free Blocks

- Method 1: *Implicit list* using lengths -- links all blocks

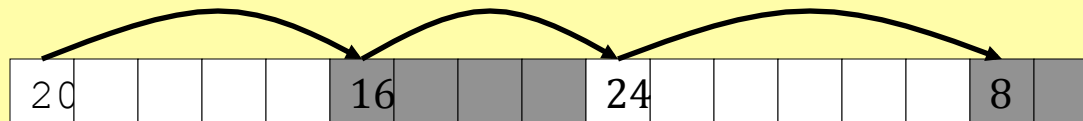


- Method 2: *Explicit list* among the free blocks using pointers within the free blocks



Keeping Track of Free Blocks

- Method 1: *Implicit list* using lengths -- links all blocks

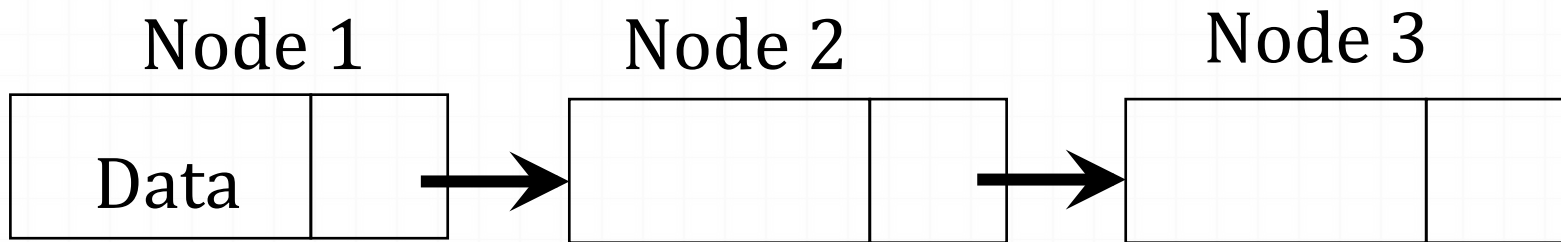


- Method 2: *Explicit list* among the free blocks using pointers within the free blocks



- Method 3: *Segregated free list*
 - Different free lists for different size classes
- Method 4: *Blocks sorted by size*
 - For example balanced tree (Red-Black?) with pointers inside each free block, block length used as key

Linked Lists



A generic linked list has a collection of node (structs)

- Each node points to the next node in the list
- Nodes located at different memory locations (unlike arrays where they are contiguous)
- Example use case: **Explicit free lists**

Advantages compared to arrays

- Nodes can be easily inserted or removed from the list without modifying the whole list

Let's look at an example of using structures, pointers, `malloc()`, and `free()` to implement a **linked list of strings**.

```
typedef struct Node node;
```

```
struct Node {  
    char *value;  
    _____ next;  
};
```

node



Q:What is the data type of the variable '**next**'?

- A. struct Node
- B. Node
- C. node
- D. node *

Let's look at an example of using structures, pointers, `malloc()`, and `free()` to implement a **linked list of strings**.

```
typedef struct Node node;
```

```
struct Node {  
    char *value;  
    _____ next;  
};
```

node



Q:What is the data type of the variable '**next**'?

- A. struct Node
- B. Node
- C. node
- D. node *

Adding a node to the list

```
node *add_node(node* head, char *string)
```

```
{  
    //Step 1:Create a new node
```

new_node

```
    _____;  
    return new_node;  
}
```

value=?

next=?

Q: How should we declare and initialize **new_node**?

A. `node *new_node=(node*) malloc(sizeof(node)) ;`

B. `node new_node ;`

C. `node *new_node=head ;`

D. `node *new_node=(node *)malloc(sizeof(head)) ;`

Adding a node to the list

```
node *add_node(node* head, char *string)
```

```
{  
    //Step 1:Create a new node
```

new_node

```
    _____;  
    return new_node;  
}
```

value=?

next=?

Q: How should we declare and initialize **new_node**?

A. `node *new_node=(node*) malloc(sizeof(node)) ;`

B. `node new_node;`

C. `node *new_node=head;`

D. `node *new_node=(node *)malloc(sizeof(head)) ;`

Q: Which of the following is true about Step 2?

```
node *list_add(node* head, char *string)
{
    //Step 1: Create a new node
    node *new_node=(node*) malloc(sizeof(node)) ;
    //Step 2: Fill in its value
    strcpy(new_node->value, string);
    return new_node;
}
```

- A. Step 2 is correct.
- B. We should use the operator '.' instead of '->'
- C. Memory is not allocated for 'value'

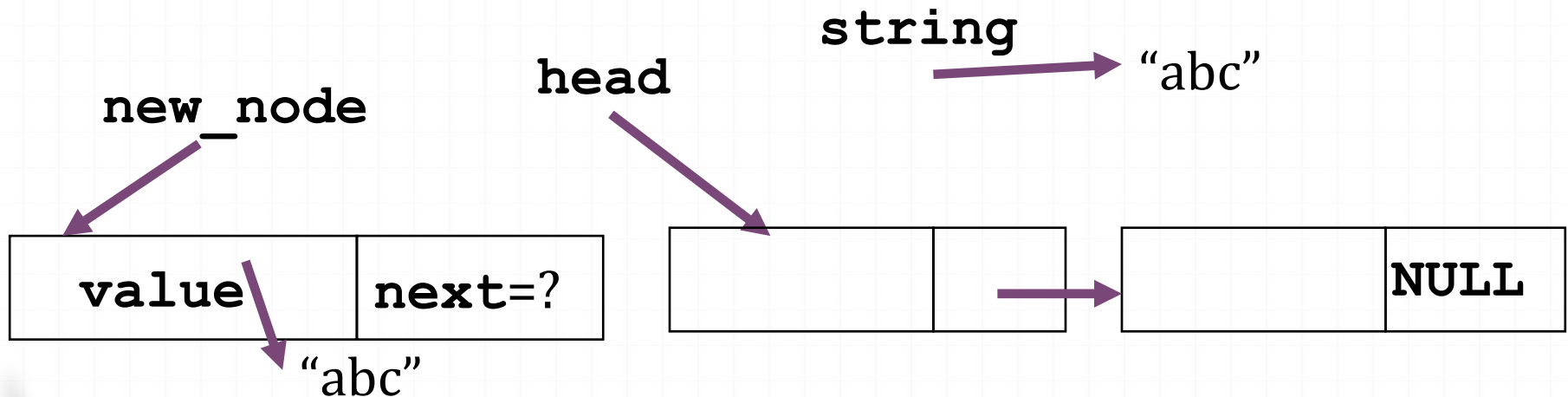
Q: Which of the following is true about Step 2?

```
node *list_add(node* head, char *string)
{
    //Step 1: Create a new node
    node *new_node=(node*) malloc(sizeof(node)) ;
    //Step 2: Fill in its value
    strcpy(new_node->value, string);
    return new_node;
}
```

- A. Step 2 is correct.
- B. We should use the operator '.' instead of '->'
- C. Memory is not allocated for 'value'

So far....

```
node *list_add(node* head, char *string)
{
    //Step 1: Create a new node
    node *new_node=(node*) malloc(sizeof(node));
    //Step 2: Fill in its value
    new_node->value = (char*) malloc(strlen(string)+1);
    strcpy(new_node->value, string);
    return new_node;
}
```



What should Step 3 be?

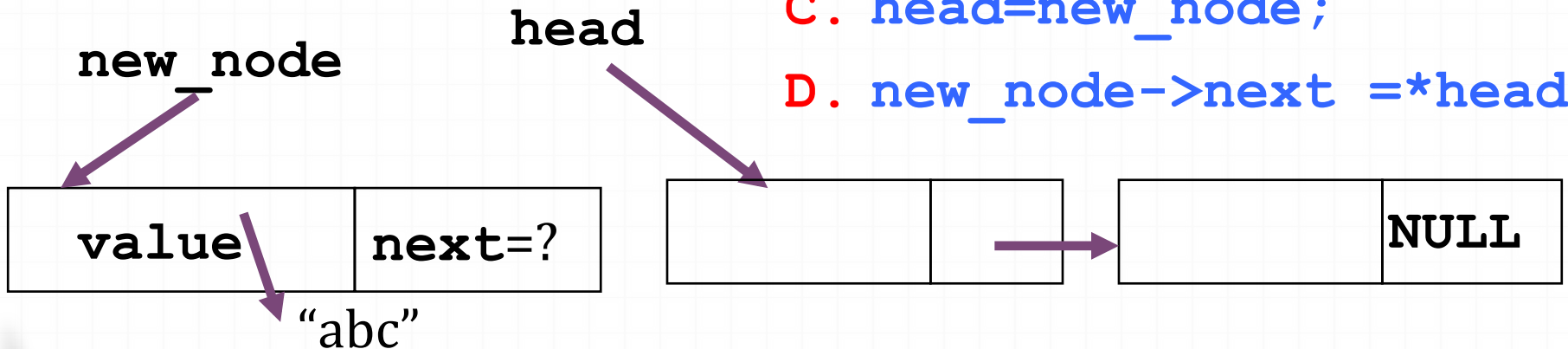
```
node *list_add(node* head, char *string){  
    //Step 1: Create a new node  
    node *new_node=(node*) malloc(sizeof(node));  
    //Step 2: Fill in its value  
    new_node->value = (char*) malloc(strlen(string)+1);  
    strcpy(new_node->value, string);  
    //Step 3:  
  
    _____  
    return new_node;  
}
```

A. `new_node->next =head;`

B. `next=head;`

C. `head=new_node;`

D. `new_node->next =*head;`



What should Step 3 be?

```
node *list_add(node* head, char *string){  
    //Step 1: Create a new node  
    node *new_node=(node*) malloc(          (node));  
    //Step 2: Fill in its value  
    new_node->value = (char*) malloc(strlen(string)+1);  
    strcpy(new_node->value, string);  
    //Step 3: Link new_node to the head of the list  
    new_node->next =head;  
    return new_node;  }
```

A. `new_node->next =head;`

B. `next=head;`

C. `head=new_node;`

D. `new_node->next =*head;`

