

Memory Management

Allocating Data in a Program

- We have seen that data can be allocated on the **stack** as part of a procedure's stack frame (e.g., local variables).
- But this data disappears once the procedure returns.
- Data can be also be allocated **statically** at compile time, making it available for the entire lifetime of the program.
- Static allocation is not really done in WLP4, but could be used to implement global variables, for example.
- But since static allocation happens at compile time, it can't depend on things like user input.

Dynamic Memory Management

- **Dynamic allocation** allows us to allocate data **at runtime** that is not tied to the scope of a procedure.
- The fact that the data needs to be allocated at runtime, and the data's lifetime is indefinite, introduces many complications.
 - How do we find free space to store the allocated data?
 - When the user is done with the data, how do we clean it up?
 - Once we clean up the data, how do we ensure the space can be reused?
 - Can we clean up the data automatically or does the user have to request it?
 - If we clean it up automatically, how do we detect that it's no longer in use?
- Different programming languages take different approaches!

Explicit & Implicit Memory Management

- Dynamic allocation stores data in a pool of memory called **the heap**.
 - No relation to the "heap data structure" you may have seen in CS 240.
- In **explicit** memory management, the programmer has a direct interface for requesting memory from the heap and returning memory back to the heap when they're done with it.
- Examples: C (malloc and free), C++/WLP4 (new and delete).
- In **implicit** memory management, when the programmer uses things like arrays or objects that require dynamic allocation, the heap management is done automatically in the background.
- Examples: Java, Racket, Python, any "garbage collected" language.

Aside: New & Delete in C++ and WLP4

- Implementing **new** and **delete** in your WLP4 code generator is mostly just a matter of calling the procedures we provide in **alloc.merl**.
- This lecture will discuss ways of implementing these procedures.
- However, there are some small catches when using these procedures:
 - **new** in C++ by default throws an exception when allocation fails. There is also a "nothrow" version of **new** that returns a null pointer.
 - In WLP4, we default to the "nothrow" version, so failed allocation should return NULL.
 - **delete** in C++ is expected to do *nothing* when you try to delete a null pointer (as opposed to crashing or causing a weird memory error).
- The provided procedures for "new" and "delete" don't work this way.

Aside: New & Delete in C++ and WLP4

- The "new" procedure provided by **alloc.merl** returns the value 0 on a failed allocation.
- In our WLP4 code generator, we use 1 for the value of NULL!
- This means you need to detect when the "new" procedure returns 0, and place 1 (for NULL) in \$3.
- The "delete" procedure provided by **alloc.merl** will crash if you tell it to delete address 1.
- So, you need to actually detect when 1 (for NULL) is about to be passed to the "delete" procedure, and *skip the procedure call*.

Lecture Overview

- We'll look at two memory management algorithms:
 - The Free List Algorithm (used in "traditional" implementations of malloc)
 - The Binary Buddy System Algorithm (used in **alloc.merl**)
- We'll discuss garbage collection algorithms, used for implicit memory management, at a high level (no implementation details).
 - Reference counting (used by `std::shared_ptr`)
 - Mark and sweep
 - Copying collectors
 - Generational garbage collection
- Modern garbage collectors generally combine several of these ideas.

The Worst Memory Allocator

- The easiest way to write an allocator is to **never free anything**.
- Choose somewhere to store your allocated data and go to town.

The Worst Memory Allocator

- The easiest way to write an allocator is to **never free anything**.
- Choose somewhere to store your allocated data and go to town.

```
a = malloc(2)
```



The Worst Memory Allocator

- The easiest way to write an allocator is to **never free anything**.
- Choose somewhere to store your allocated data and go to town.

```
a = malloc(2)
```

```
b = malloc(4)
```



The Worst Memory Allocator

- The easiest way to write an allocator is to **never free anything**.
- Choose somewhere to store your allocated data and go to town.

```
a = malloc(2)
b = malloc(4)
c = malloc(1)
```



The Worst Memory Allocator

- The easiest way to write an allocator is to **never free anything**.
- Choose somewhere to store your allocated data and go to town.

```
a = malloc(2)
b = malloc(4)
c = malloc(1)
```



- This works perfectly for small toy programs!
- But it's obviously not suitable for real-world use....

A Slightly Better Allocator

- Freeing is relatively easy if all blocks of memory are **the same size**.
- The idea is to maintain a **linked list of free blocks**.

A Slightly Better Allocator

- Freeing is relatively easy if all blocks of memory are **the same size**.
- The idea is to maintain a **linked list of free blocks**.

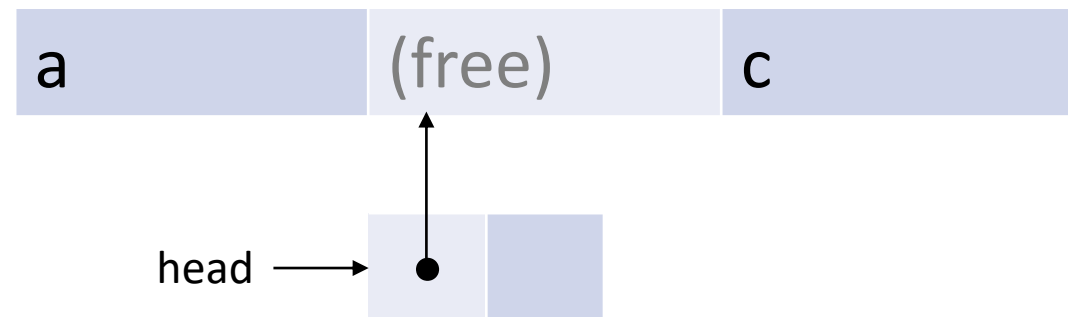
```
a = malloc()  
b = malloc()  
c = malloc()
```



A Slightly Better Allocator

- Freeing is relatively easy if all blocks of memory are **the same size**.
- The idea is to maintain a **linked list of free blocks**.

```
a = malloc() free(b)  
b = malloc()  
c = malloc()
```

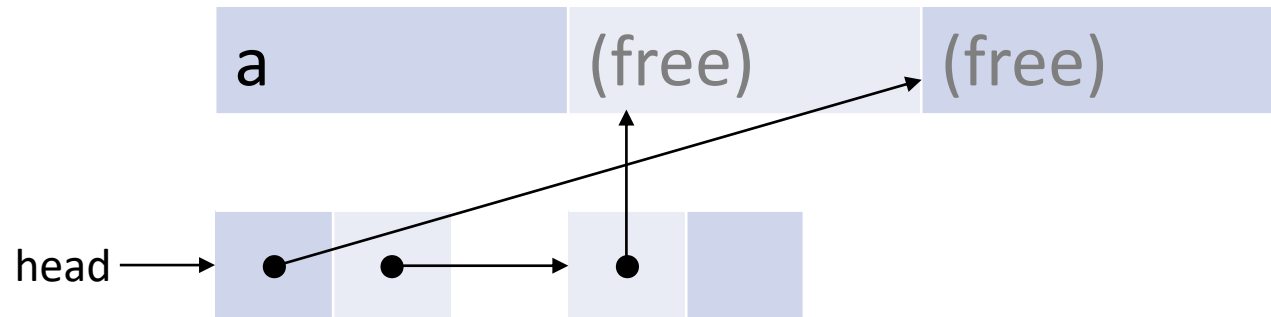


- When freeing, add the block to the front of the free list.

A Slightly Better Allocator

- Freeing is relatively easy if all blocks of memory are **the same size**.
- The idea is to maintain a **linked list of free blocks**.

```
a = malloc() free(b)
b = malloc() free(c)
c = malloc()
```

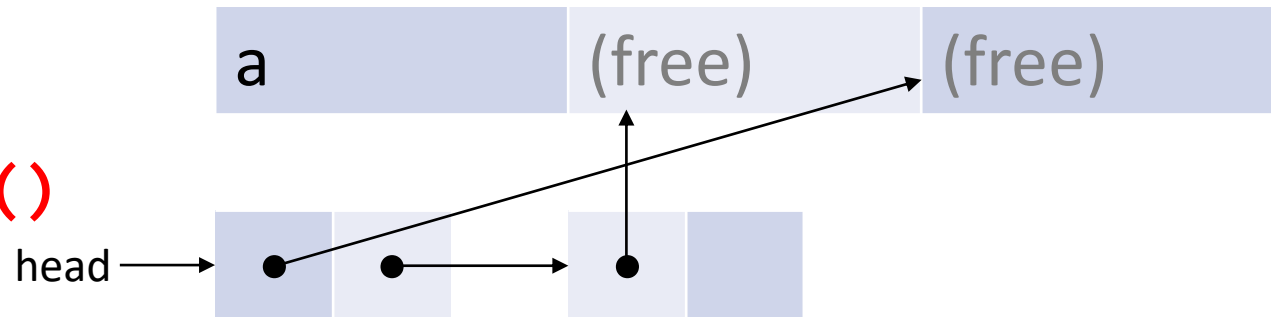


- When freeing, add the block to the front of the free list.

A Slightly Better Allocator

- Freeing is relatively easy if all blocks of memory are **the same size**.
- The idea is to maintain a **linked list of free blocks**.

```
a = malloc() free(b)  
b = malloc() free(c)  
c = malloc() d = malloc()
```

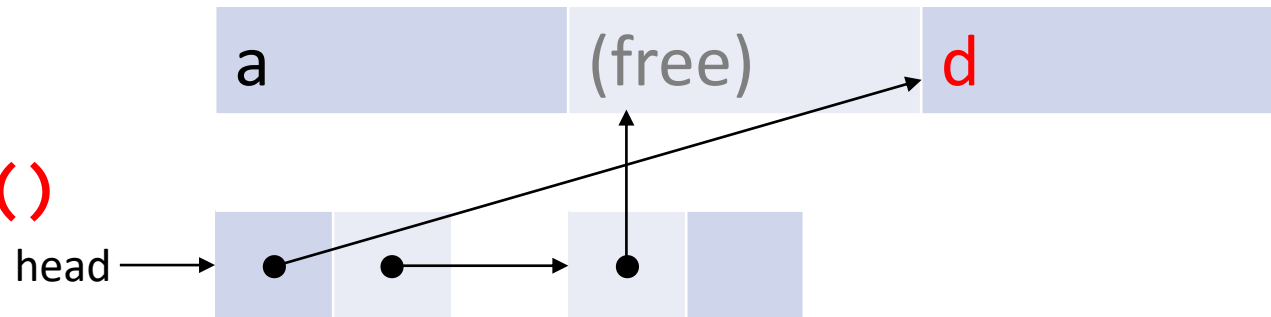


- When freeing, add the block to the front of the free list.

A Slightly Better Allocator

- Freeing is relatively easy if all blocks of memory are **the same size**.
- The idea is to maintain a **linked list of free blocks**.

```
a = malloc() free(b)  
b = malloc() free(c)  
c = malloc() d = malloc()
```

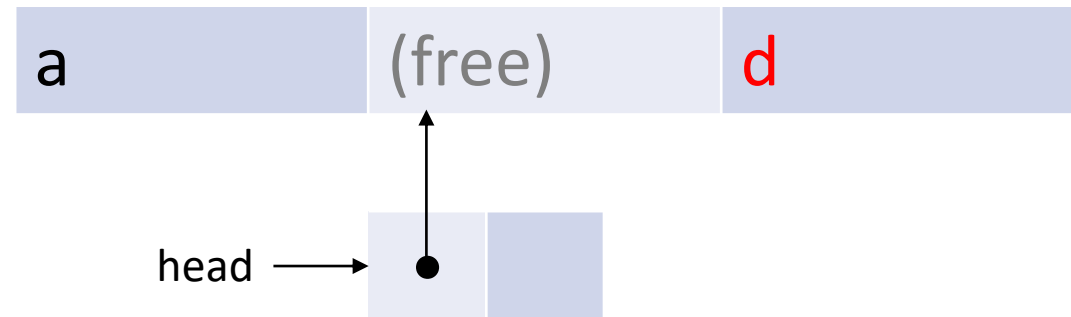


- When freeing, add the block to the front of the free list.

A Slightly Better Allocator

- Freeing is relatively easy if all blocks of memory are **the same size**.
- The idea is to maintain a **linked list of free blocks**.

```
a = malloc() free(b)
b = malloc() free(c)
c = malloc() d = malloc()
```

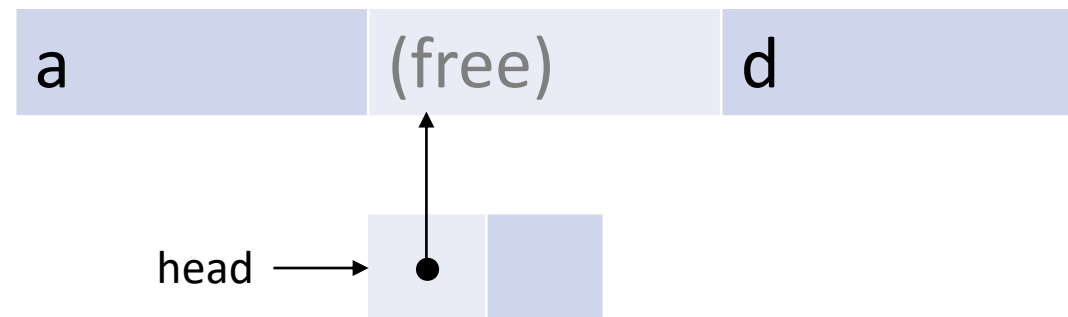


- When freeing, add the block to the front of the free list.

A Slightly Better Allocator

- Freeing is relatively easy if all blocks of memory are **the same size**.
- The idea is to maintain a **linked list of free blocks**.

```
a = malloc() free(b)
b = malloc() free(c)
c = malloc() d = malloc()
```

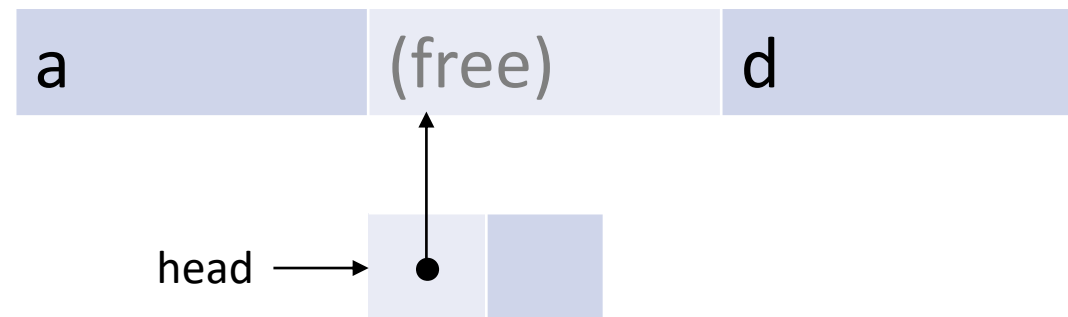


- When freeing, add the block to the front of the free list.

A Slightly Better Allocator

- Freeing is relatively easy if all blocks of memory are **the same size**.
- The idea is to maintain a **linked list of free blocks**.

```
a = malloc() free(b)
b = malloc() free(c)
c = malloc() d = malloc()
```



- Two cases when allocating:
 - If there's a free block available, reuse it (remove it from the front of the list).
 - Otherwise, allocate at the front of "unused heap space".
- When freeing, add the block to the front of the free list.

Supporting Variable Size Blocks

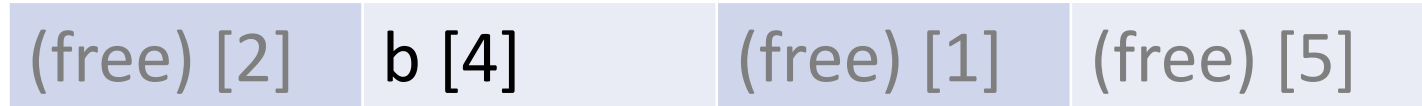
- Why does using fixed size blocks make things easier?
 - We can always take the **first block** in the free list (it will be the right size).
 - No possibility of **fragmentation**: when the heap has enough free space *in total* to support an allocation, but not enough space in individual blocks.

```
a = malloc(2) free(a)
```

```
b = malloc(4) free(c)
```

```
c = malloc(1) free(d)
```

```
d = malloc(5) e = malloc(6) *Not to scale
```



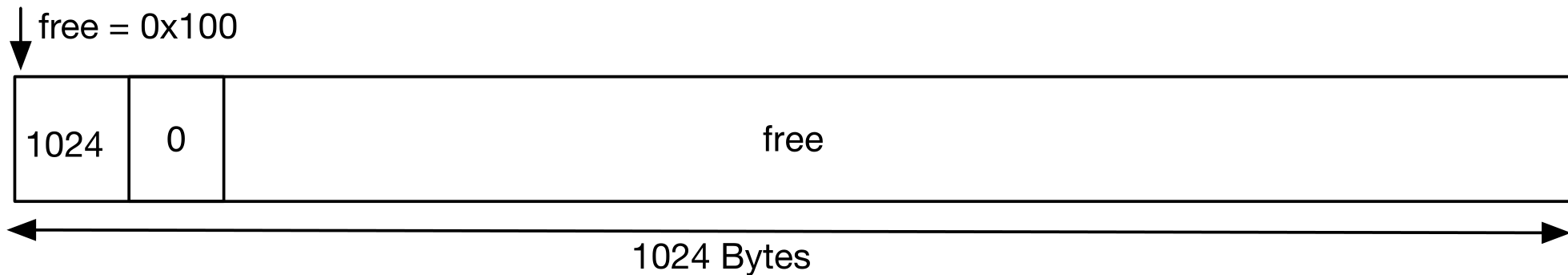
- In this example, the first block is too small, so we need to look further.
- The second and third block are also too small. But if we could **merge** these blocks there would be enough space.

The Free List Algorithm

- **Idea:** Maintain a linked list of (variable size) free blocks, but perform merging when we have adjacent free blocks.
- If the free list is unsorted, it's hard to (efficiently) tell when two blocks are beside each other.
- We'll maintain the free list in **sorted order by block address**.
- Maintaining the order takes a little bit of extra time but makes merging efficient and simple.
- But how do we store linked list nodes if we don't have a way to dynamically allocate memory yet??
- We can store the links between free blocks **in the blocks themselves**.

The Free List Algorithm: Example

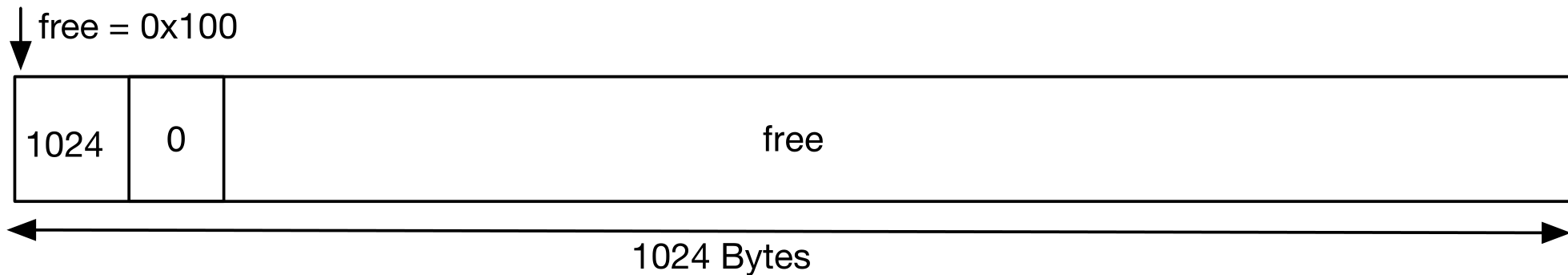
- Each block is a sequence of **words** (think of it as an array).
- The first word in the block stores the **size** in bytes.
- **If the block is free**, the second word is the **address of the next free block** in the free list.



The Free List Algorithm: Example

- Each block is a sequence of **words** (think of it as an array).
- The first word in the block stores the **size** in bytes.

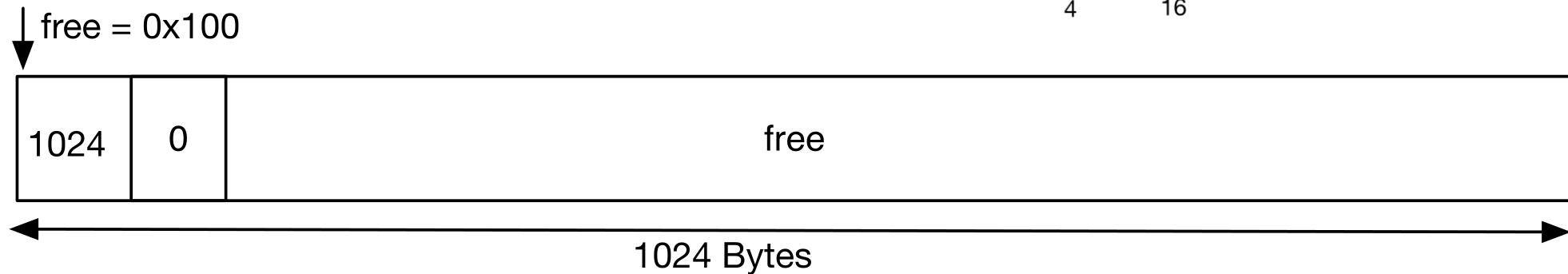
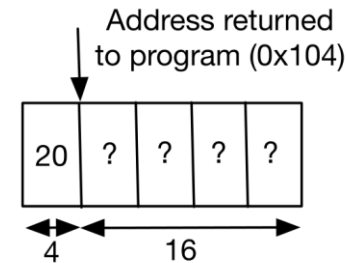
A = malloc(16)



The Free List Algorithm: Example

- Each block is a sequence of **words** (think of it as an array).
- The first word in the block stores the **size** in bytes.
- We actually allocate **20 bytes** because we need one extra word to store the size of the block.

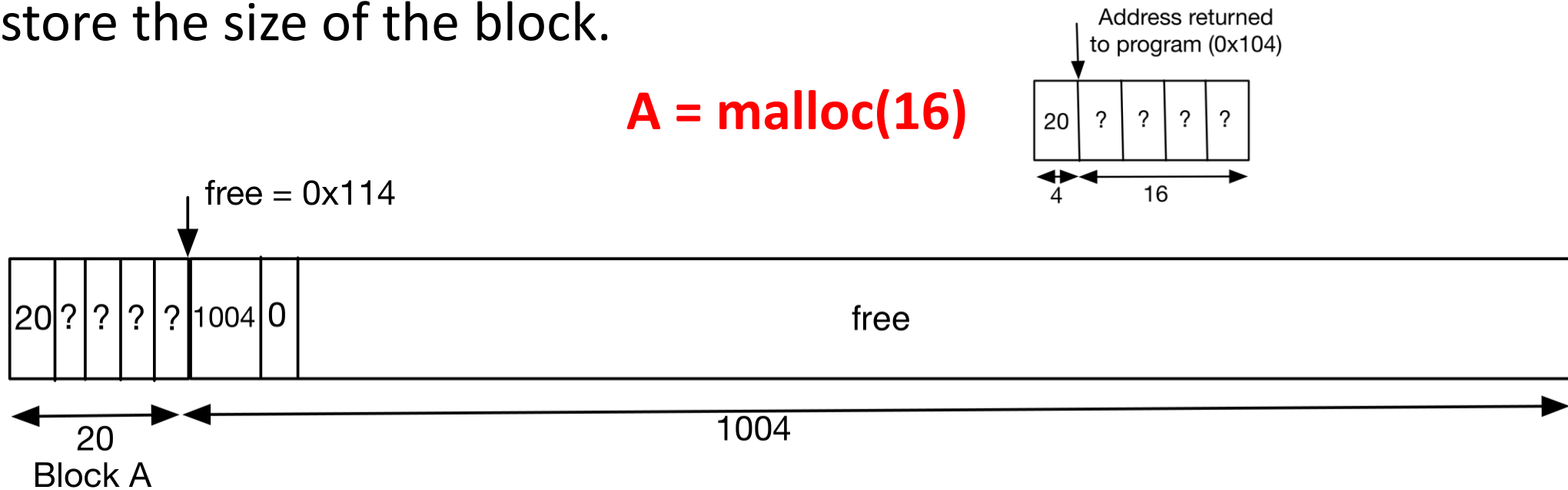
A = malloc(16)



The Free List Algorithm: Example

- Each block is a sequence of **words** (think of it as an array).
- The first word in the block stores the **size** in bytes.
- We actually allocate **20 bytes** because we need one extra word to store the size of the block.

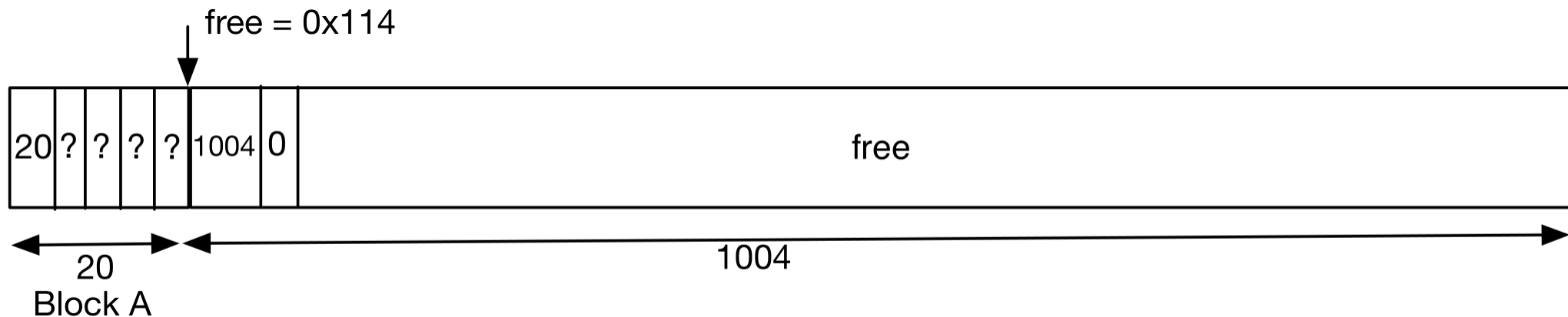
A = malloc(16)



The Free List Algorithm: Example

- Each block is a sequence of **words** (think of it as an array).
- The first word in the block stores the **size** in bytes.
- We actually allocate **20 bytes** because we need one extra word to store the size of the block.

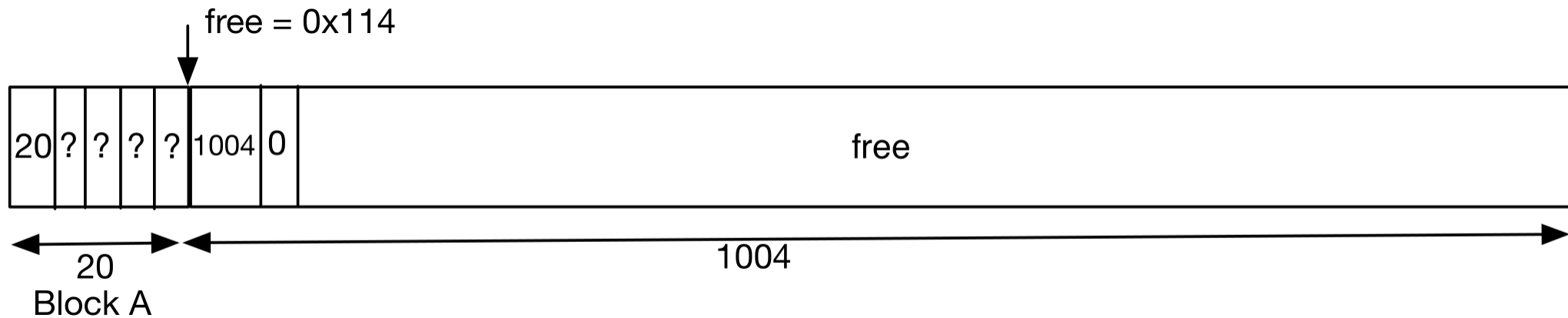
A = malloc(16)



The Free List Algorithm: Example

- Each block is a sequence of **words** (think of it as an array).
- The first word in the block stores the **size** in bytes.

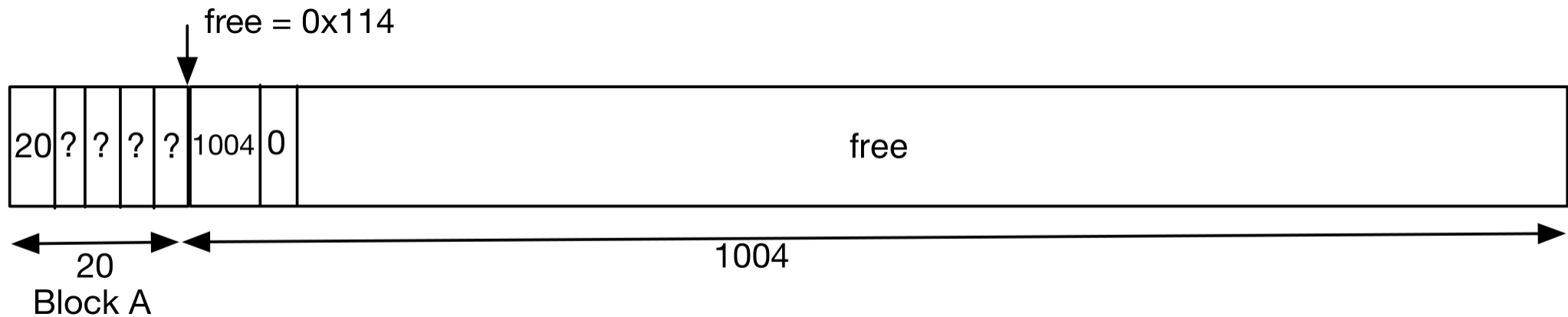
B = malloc(28)



The Free List Algorithm: Example

- Each block is a sequence of **words** (think of it as an array).
- The first word in the block stores the **size** in bytes.
- We actually allocate **32 bytes**.

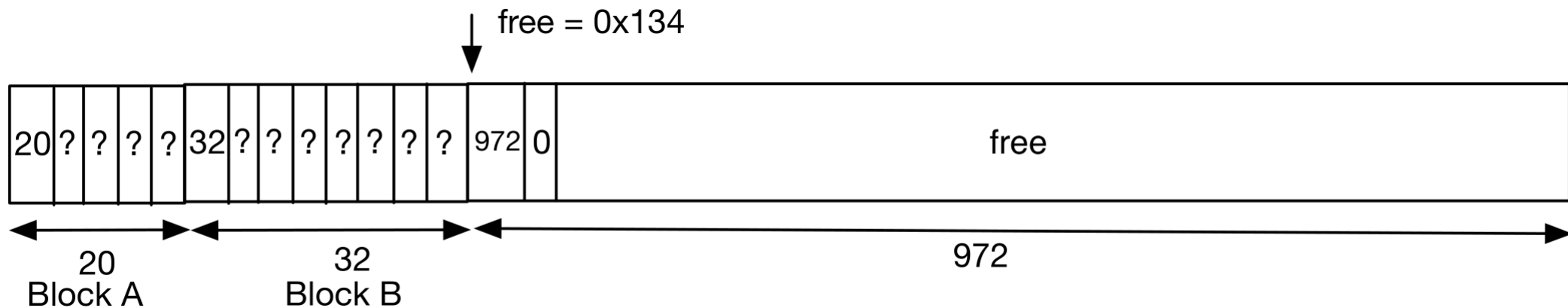
B = malloc(28)



The Free List Algorithm: Example

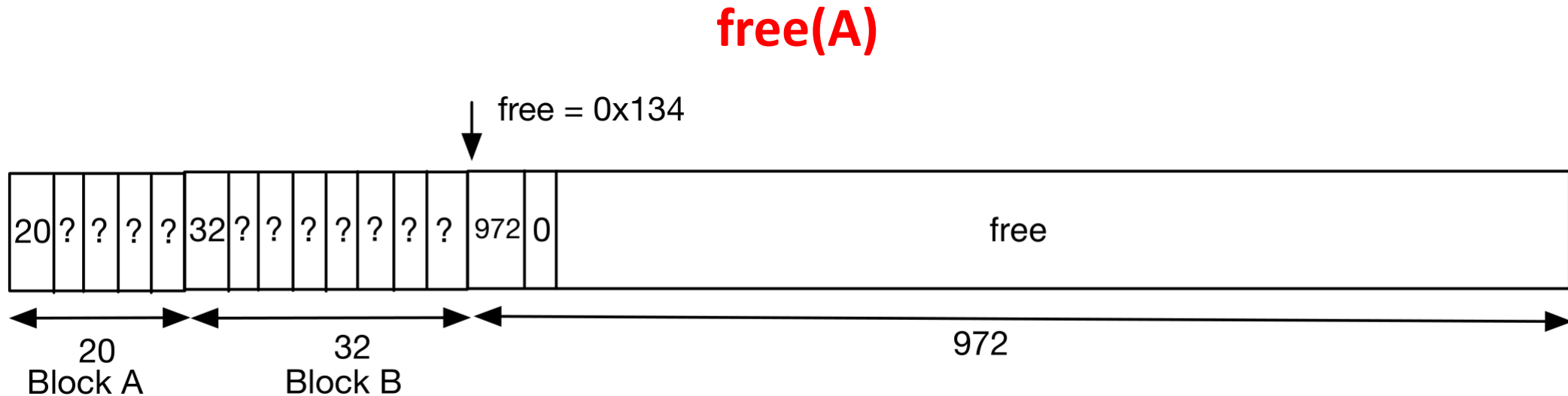
- Each block is a sequence of **words** (think of it as an array).
- The first word in the block stores the **size** in bytes.
- We actually allocate **32 bytes**.

B = malloc(28)



The Free List Algorithm: Example

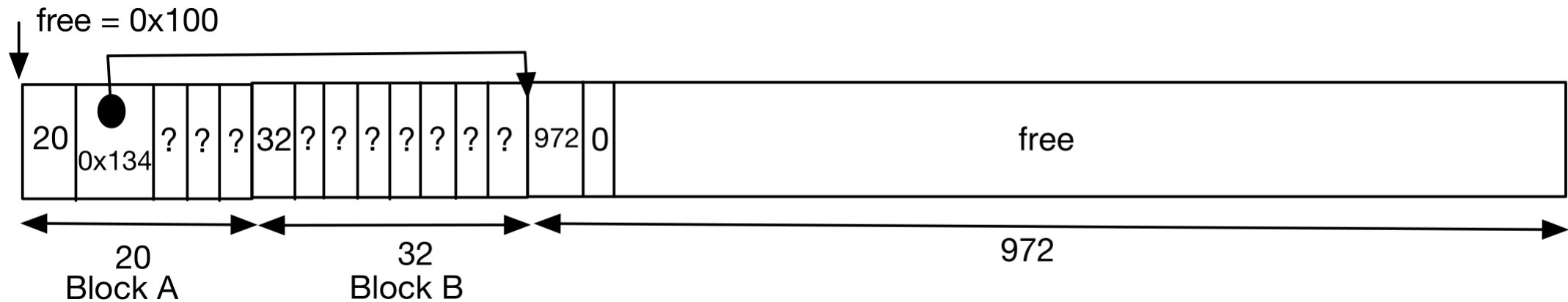
- Each block is a sequence of **words** (think of it as an array).
- The first word in the block stores the **size** in bytes.



The Free List Algorithm: Example

- Each block is a sequence of **words** (think of it as an array).
- The first word in the block stores the **size** in bytes.

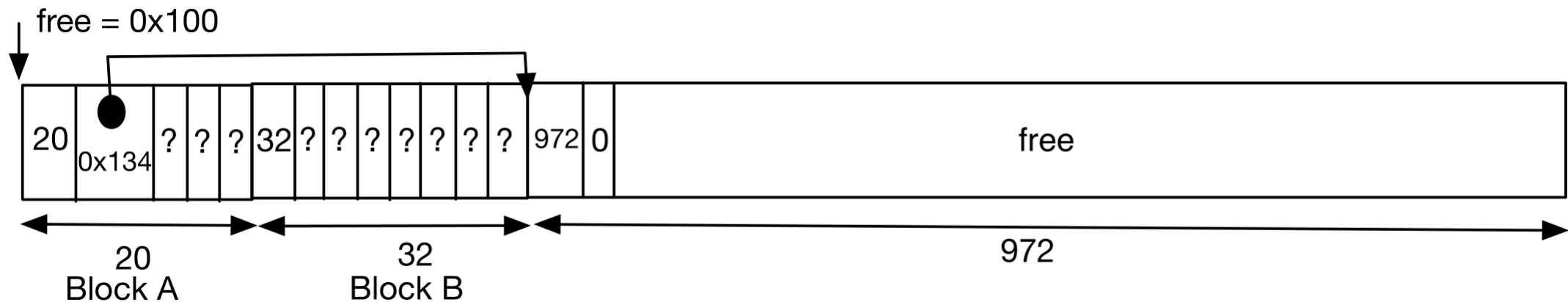
free(A)



The Free List Algorithm: Example

- Each block is a sequence of **words** (think of it as an array).
- The first word in the block stores the **size** in bytes.
- **If the block is free**, the second word is the **address of the next free block** in the free list.

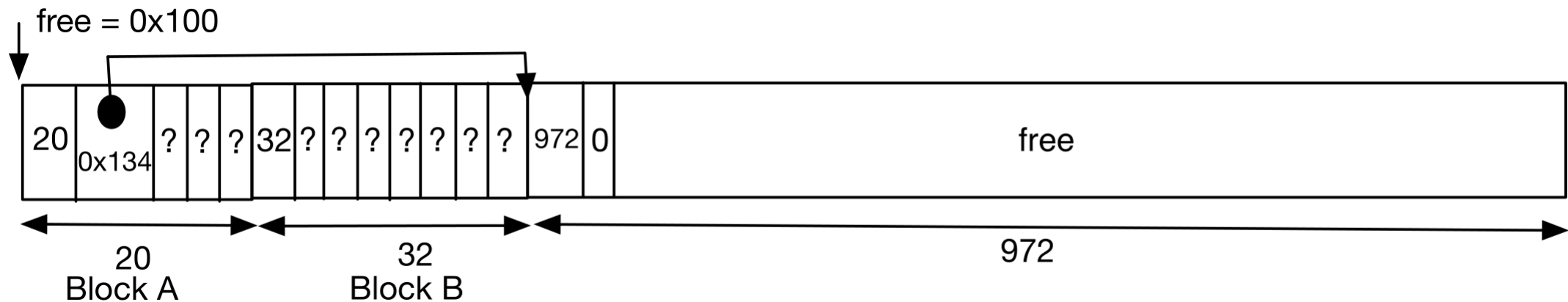
free(A)



The Free List Algorithm: Example

- Each block is a sequence of **words** (think of it as an array).
- The first word in the block stores the **size** in bytes.
- **If the block is free**, the second word is the **address of the next free block** in the free list.

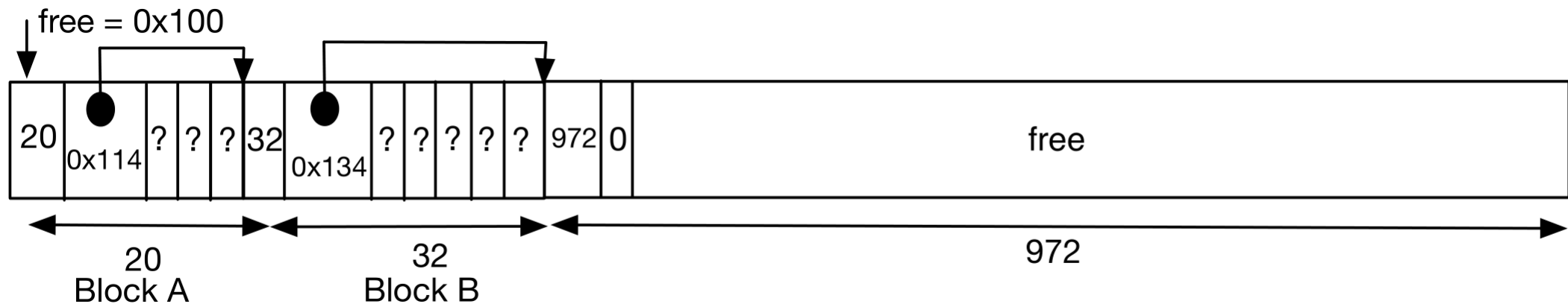
free(B)



The Free List Algorithm: Example

- Each block is a sequence of **words** (think of it as an array).
- The first word in the block stores the **size** in bytes.
- **If the block is free**, the second word is the **address of the next free block** in the free list.

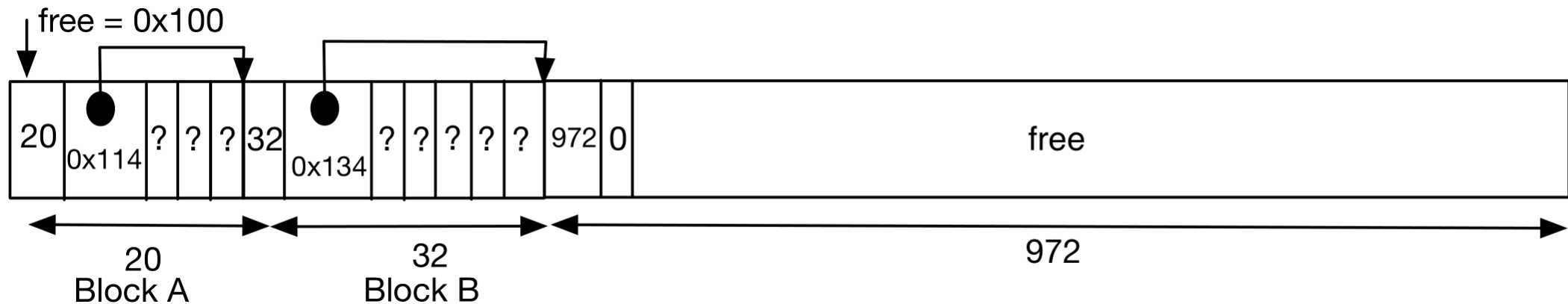
free(B)



The Free List Algorithm: Example

- Each block is a sequence of **words** (think of it as an array).
- The first word in the block stores the **size** in bytes.
- We're not done! We have **adjacent free blocks**, so they can be merged.

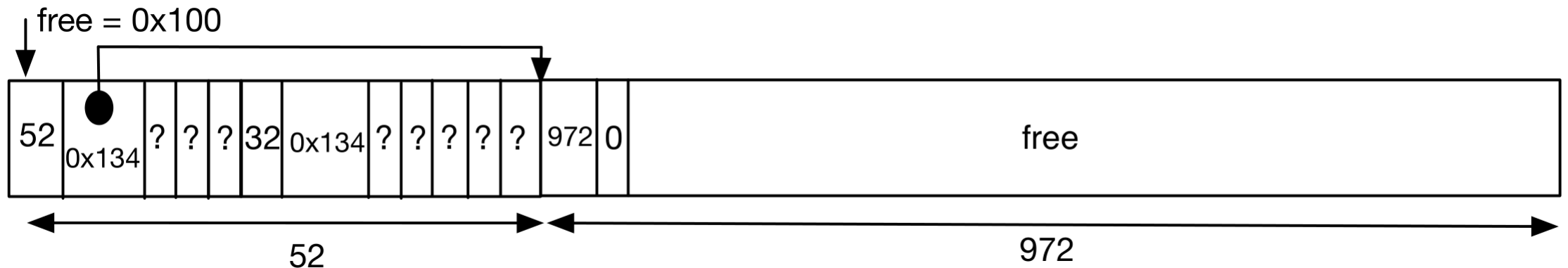
free(B)



The Free List Algorithm: Example

- Each block is a sequence of **words** (think of it as an array).
- The first word in the block stores the **size** in bytes.
- We're not done! We have **adjacent free blocks**, so they can be merged. **Merge the two left blocks...**

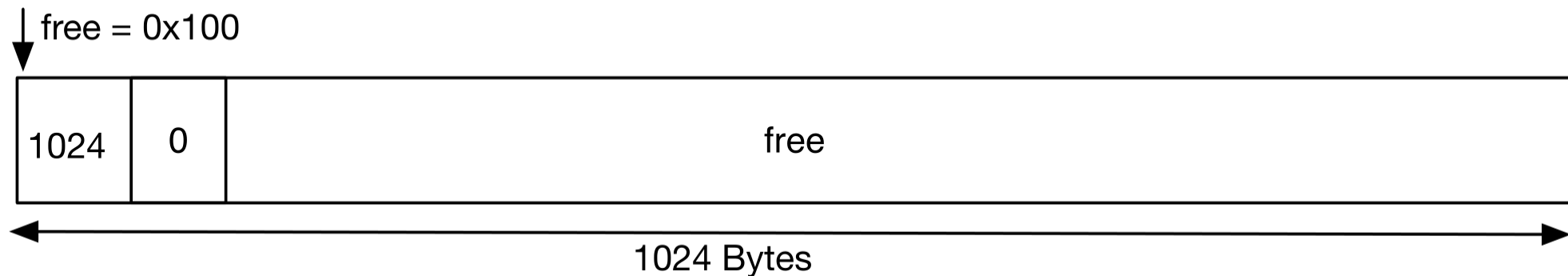
free(B)



The Free List Algorithm: Example

- Each block is a sequence of **words** (think of it as an array).
- The first word in the block stores the **size** in bytes.
- We're not done! We have **adjacent free blocks**, so they can be merged. **Then merge with the block to the right...**

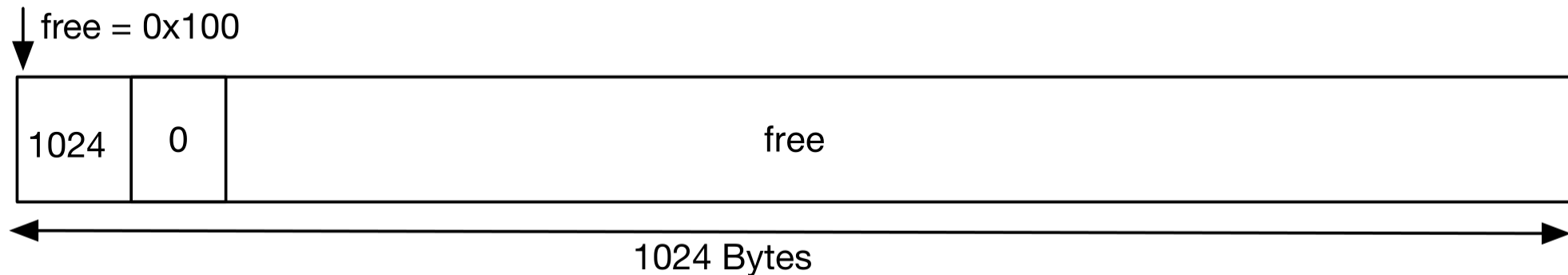
free(B)



The Free List Algorithm: Example

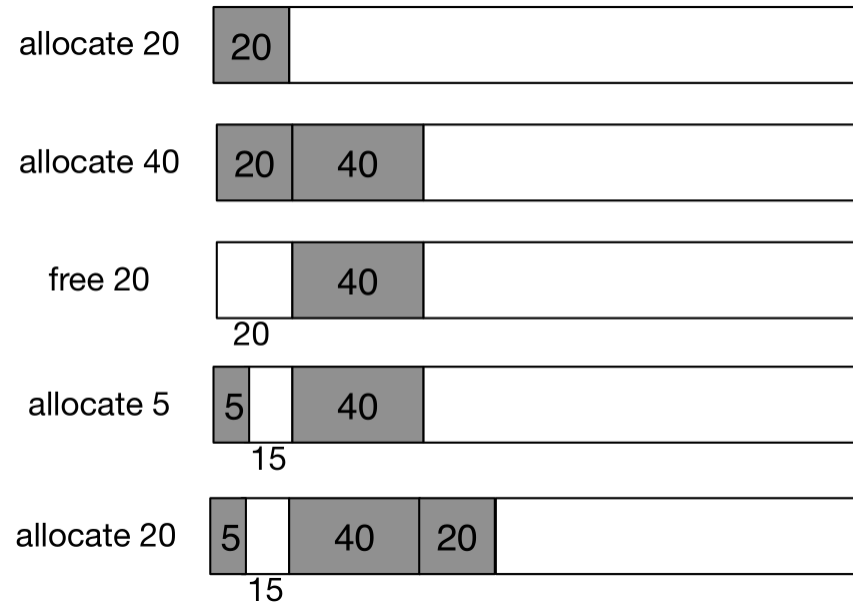
- Each block is a sequence of **words** (think of it as an array).
- The first word in the block stores the **size** in bytes.
- Both blocks we allocated were freed, and the heap is back in its initial state.

free(B)



Problems with the Free List Approach

- Merging reduces fragmentation, but it can still happen.



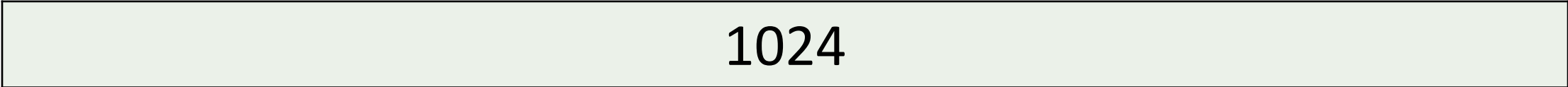
Problems with the Free List Approach

- Merging reduces fragmentation, but it can still happen.
- Different heuristics for finding a free block can affect fragmentation:
 - First fit: Find the **first** block in the free list that fits and allocate there. Fast but makes no attempt to reduce fragmentation.
 - Best fit: Find the free block whose size is **closest** to the amount we want to allocate. Slower but leaves smaller "gaps" behind.
 - Worst fit: Find the free block whose size is **farthest** from the amount we want to allocate. Slower but leaves larger "gaps" behind.
- None is strictly better than the others (in a theoretical sense). It depends on the patterns of allocation.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To allocate, we recursively split the heap in half until we have a block whose size is the **smallest power of 2** needed to store the data.

malloc(24) (need 28 bytes, smallest power of 2 is 32)



1024

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To allocate, we recursively split the heap in half until we have a block whose size is the **smallest power of 2** needed to store the data.

malloc(24) (need 28 bytes, smallest power of 2 is 32)



- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To allocate, we recursively split the heap in half until we have a block whose size is the **smallest power of 2** needed to store the data.

malloc(24) (need 28 bytes, smallest power of 2 is 32)



- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To allocate, we recursively split the heap in half until we have a block whose size is the **smallest power of 2** needed to store the data.

malloc(24) (need 28 bytes, smallest power of 2 is 32)

128	128	256	512
-----	-----	-----	-----

- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To allocate, we recursively split the heap in half until we have a block whose size is the **smallest power of 2** needed to store the data.

malloc(24) (need 28 bytes, smallest power of 2 is 32)

64	64	128	256	512
----	----	-----	-----	-----

- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To allocate, we recursively split the heap in half until we have a block whose size is the **smallest power of 2** needed to store the data.

malloc(24) (need 28 bytes, smallest power of 2 is 32)



- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To allocate, we recursively split the heap in half until we have a block whose size is the **smallest power of 2** needed to store the data.

malloc(24) (need 28 bytes, smallest power of 2 is 32)



- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To allocate, we recursively split the heap in half until we have a block whose size is the **smallest power of 2** needed to store the data.

malloc(40) (need 44 bytes, smallest power of 2 is 64)



- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To allocate, we recursively split the heap in half until we have a block whose size is the **smallest power of 2** needed to store the data.

malloc(40) (need 44 bytes, smallest power of 2 is 64)



- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To allocate, we recursively split the heap in half until we have a block whose size is the **smallest power of 2** needed to store the data.

malloc(50) (need 54 bytes, smallest power of 2 is 64)



- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To allocate, we recursively split the heap in half until we have a block whose size is the **smallest power of 2** needed to store the data.

malloc(50) (need 54 bytes, smallest power of 2 is 64)



- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To allocate, we recursively split the heap in half until we have a block whose size is the **smallest power of 2** needed to store the data.

malloc(50) (need 54 bytes, smallest power of 2 is 64)



- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To free a block, we first mark it as free. If its **buddy** is also free, we merge. Continue merging until no more merges are possible.



- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To free a block, we first mark it as free. If its **buddy** is also free, we merge. Continue merging until no more merges are possible.

Free middle block (first 64 byte block)



- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To free a block, we first mark it as free. If its **buddy** is also free, we merge. Continue merging until no more merges are possible.

Free middle block (first 64 byte block)



- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To free a block, we first mark it as free. If its **buddy** is also free, we merge. Continue merging until no more merges are possible.

Free leftmost block (32 bytes)



- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To free a block, we first mark it as free. If its **buddy** is also free, we merge. Continue merging until no more merges are possible.

Free leftmost block (32 bytes)



- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To free a block, we first mark it as free. If its **buddy** is also free, we merge. Continue merging until no more merges are possible.

Free leftmost block (32 bytes) Merge...



- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To free a block, we first mark it as free. If its **buddy** is also free, we merge. Continue merging until no more merges are possible.

Free leftmost block (32 bytes) Merge... Merge again...



- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To free a block, we first mark it as free. If its **buddy** is also free, we merge. Continue merging until no more merges are possible.

Free leftmost block (32 bytes) Merge... Merge again... Done.



- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To free a block, we first mark it as free. If its **buddy** is also free, we merge. Continue merging until no more merges are possible.

Free last block



- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To free a block, we first mark it as free. If its **buddy** is also free, we merge. Continue merging until no more merges are possible.

Free last block



- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To free a block, we first mark it as free. If its **buddy** is also free, we merge. Continue merging until no more merges are possible.

Free last block Merge...



- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To free a block, we first mark it as free. If its **buddy** is also free, we merge. Continue merging until no more merges are possible.

Free last block Merge... Merge again...



- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To free a block, we first mark it as free. If its **buddy** is also free, we merge. Continue merging until no more merges are possible.

Free last block Merge... Merge again... Merge again...

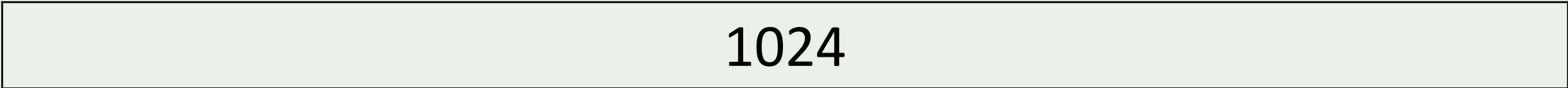


- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To free a block, we first mark it as free. If its **buddy** is also free, we merge. Continue merging until no more merges are possible.

Free last block Merge... Merge again... Merge again... Merge again...



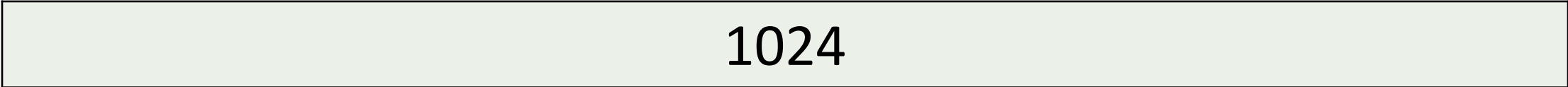
1024

- When we split a block, the two halves are called **buddies**.

The Binary Buddy System Algorithm

- This algorithm uses a fairly different approach: **splitting the heap**.
- The heap size begins as a power of 2.
- To free a block, we first mark it as free. If its **buddy** is also free, we merge. Continue merging until no more merges are possible.

The heap is restored to its original state.



1024

- When we split a block, the two halves are called **buddies**.

Binary Buddy Bookkeeping

- Like with the free list algorithm, we reserve one word in each block to keep track of the size.
- But we don't simply store the number of bytes. We store a **code**:
 - The initial block has code 1.
 - When we split the block, the left half gets code 10 and the right gets code 11.
 - If we split the left block, the codes would be 100 and 101. And so on.
- If a block's code has n digits, and the heap has size S , the size of the block is $S/2^{n-1}$.
- We can store the codes of free blocks in a list.
 - Codes can be used to determine block **addresses** as well as sizes.
 - If we flip the last bit of the block's code, we get the code for its buddy.

Garbage Collection

- Implicit memory management relies on a process called **garbage collection**: automatic reclaiming of memory that is no longer in use.

```
void typicalJavaCode() {  
    AbstractSingletonProxyFactoryBean bean =  
        new AbstractSingletonProxyFactoryBean();  
} // bean is no longer accessible
```

- After the typicalJavaCode function returns, the memory used for the AbstractSingletonProxyFactoryBean can be reclaimed.
- How do we know that we no longer need our bean?

Garbage Collection

```
Bean moreJavaCode(Bean a, Bean b) {  
    BeanFactory beanFactory = null;  
    if(a.tastierThan(b)) {  
        BeanFactory temp = new BeanFactory();  
        beanFactory = temp;  
    } // temp goes out of scope  
    Bean magicBean = beanFactory.buildBean();  
    ...  
} // beanFactory is no longer accessible
```

- In this example, "temp" goes out of scope, but is still accessible through the variable "beanFactory", so it can't be garbage collected!

Reference Counting

- Idea: Keep an internal counter of the number of pointers to each block (called the **reference count** for the block).
- When (and only when) the reference count is 0, the block can be deallocated.
- This is the idea behind `std::shared_ptr` in C++.
- Fairly cheap computationally (pointer operations become slightly more expensive since the reference count needs to be updated).
- However, the algorithm can't (easily) handle **circular references** since the reference count will never become zero.

Mark and Sweep

- This is known as a "Stop the World" algorithm because the program has to pause execution until the algorithm finishes.
- In the "Mark" phase, first the stack and global data are scanned for pointers into the heap.
- The blocks found are marked as reachable. Then the reachable blocks are scanned for pointers to more reachable blocks (and so on...)
- In the "Sweep" phase, all blocks not marked as reachable during the "Mark" phase are deallocated.
- Does not have problems with circular references, but long pauses or unpredictable pauses are unacceptable in some kinds of programs.

Copying Collector

- The heap is split into two halves named **from** and **to**.
- Memory is only allocated in the **from** half.
- When the **from** half is full, find all the reachable blocks (using a similar process to Mark and Sweep) and copy the reachable blocks to the **to** half. Then the roles of **from** and **to** are reversed.
- The heap can be **compacted** when copying, avoiding fragmentation.
- Downsides:
 - It's a "Stop the World" algorithm.
 - The amount of heap memory available is cut in half.

Generational Garbage Collection

- Copying collectors work well when **few** objects survive collection, while mark-and-sweep works well when **most** objects survive.
- Most objects in a program "die young"; they are only used for a short time and then must be deallocated.
- Divide the heap into "generations" based on the "age" of the objects in the heap.
- Use different techniques for different generations, e.g., copying for younger generations and mark-and-sweep for older generations.
- Frequency of garbage collection can vary by generation as well.