Allocator Design Principles

Allocators are designed to facilitate the management of memory that is dynamically allocated. The basic specification, in the form of the application programming interface (API), must be supplemented with practical constraints:

1. The allocator must accommodate an arbitrary sequence of malloc and free requests.

2. free must be used only to deallocate a block that was allocated previously with a call to malloc.

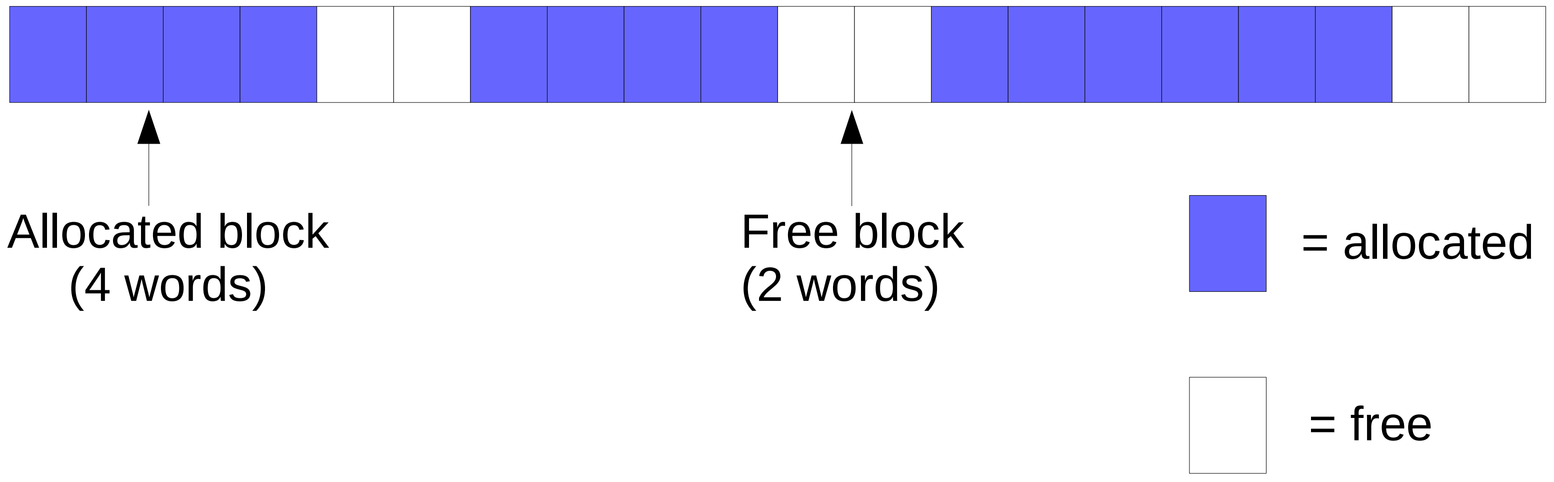
3. malloc must honor the size requested by the programmer or simply return an error.

Allocator Design Principles (cont.)

4. malloc must not access or otherwise interfere with the use of allocated blocks. This constraint includes a prohibition on moving an allocated block.

5. Hardware architecture demands that blocks of memory be aligned on double word boundaries for performance optimization. Hence, on 64 bit systems, blocks of memory must be aligned on 16 byte boundaries, i.e., must begin on a multiple of 16. On 32 bit systems, blocks must be aligned on 8 byte boundaries. Although 32 bit systems are now uncommon, a great deal of documentation about memory management still uses examples that are based on 32 bit systems.

Graphic example assumption:
For purposes of illustration, each rectangle represents one word of computer memory. On 64-bit systems, the word size is 64 bits, or 8 bytes.

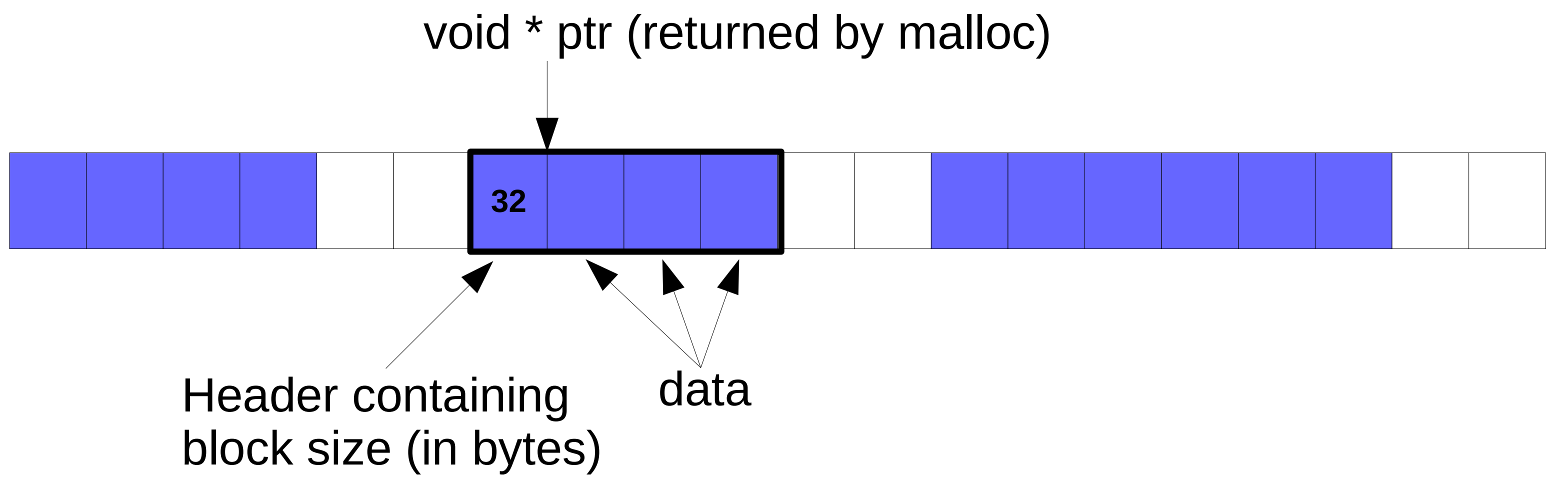


In order to manage memory, we must know at least two characteristics of each block:

1. the size of the block, and
2. whether the block is allocated or free.

How can we maintain this information efficiently?

1. We can store the size of the block at its beginning. Hence, the first word of a block of memory will become a header.
The allocator (malloc) will return the address of the second word in the block, where the programmer can safely store data, thereby preventing programmer errors.


void * ptr (returned by malloc)
32
Header containing block size (in bytes)
data

2. It would be a waste of space to use one or more bytes to indicate whether the block is allocated or free. All we really need to represent this state is a single bit; a 1 would indicate the block is allocated and a 0 would indicate that it is free.

2. (cont.)
Serendipitously however, the byte alignment constraint results in the final four bits of the block’s address being set to zeroes. Why? Note that in hexadecimal, the value 16, written as a single byte, is 0x10. Thus, the second digit represents a nybble containing four zero bits. Any multiple of 16 will always end in the same nybble.

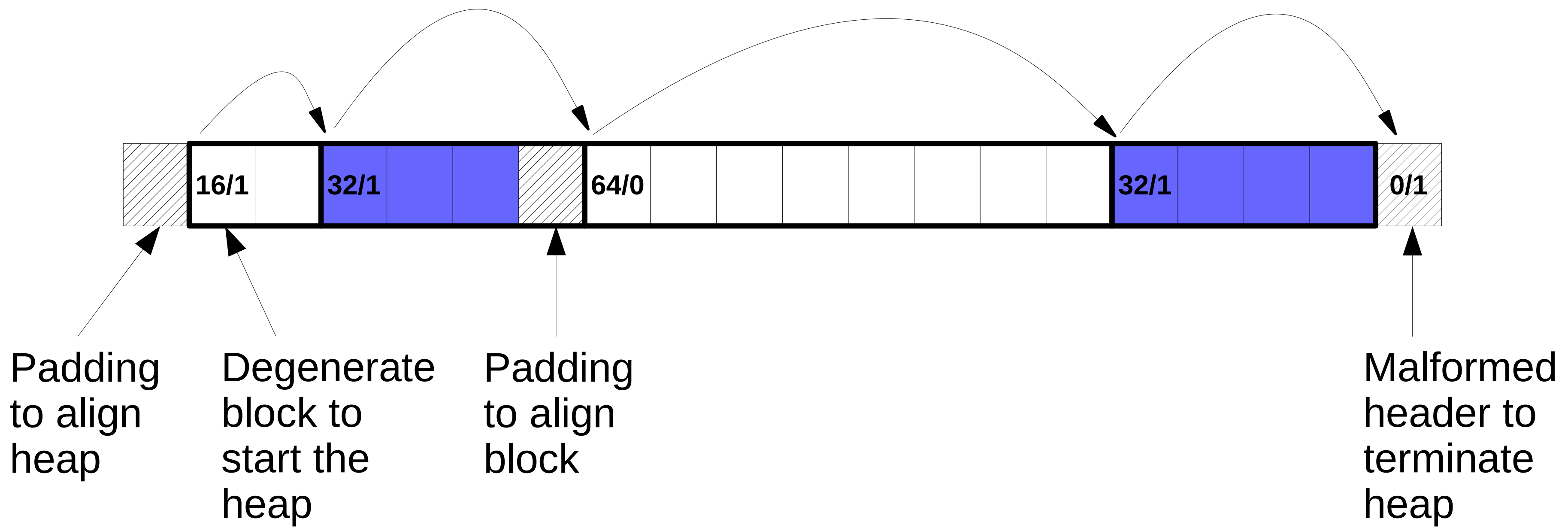
Because these bits are unessential, we can store data in them, provided that we remove the data when computing the size of the block.

Consequently, we store the state of the block in the final bit of the header. Hence, the header value will contain both the size of the block and a flag as to whether the block is allocated or free:

State = header value & 1

Size = header value & -2
(Note: -2 in binary is all 1’s with 0 at the end)

Using size information from the header, we can traverse the entire heap as an implicit list:


16/1
32/1
64/0
32/1
0/1
Padding to align heap
Degenerate block to start the heap
Padding to align block
Malformed header to terminate heap

Finding a free block

The simplest means for finding a free block to allocate is to traverse the list until a block is found that 1) is free and 2) is large enough to satisfy the request. This approach is known as “First Fit.”

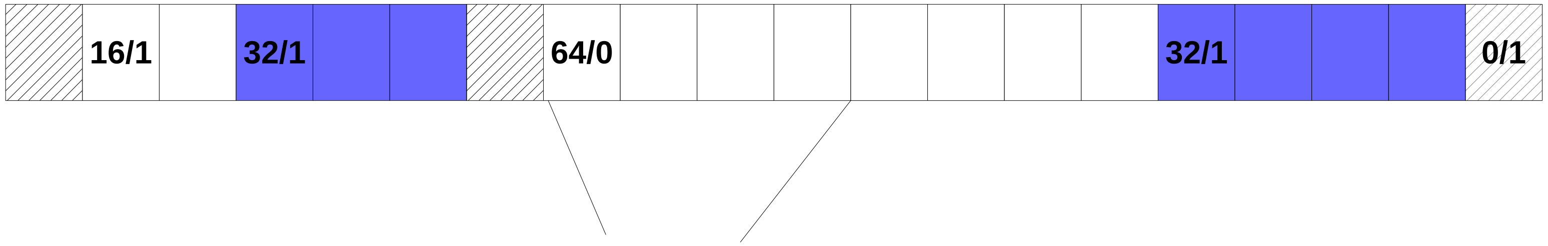
First Fit sample pseudocode:

```
bp= first_block_of_heap // pointer to header
while(
  (bp < end_of_heap)   // still within heap
  && ( (*bp & 1)   // allocated
  || (*bp <= requested_size) ) ) // too small
       bp= bp + (*bp & -2)  // try next block
if ( bp < end_of_heap )
       // free block found! Address to return is
       // bp + sizeof(char*)
```

Internal fragmentation

If we return the block as soon as we find it, we might be returning a block that is substantially larger than the size requested by the programmer in the call to malloc. This unused space is known as internal fragmentation.

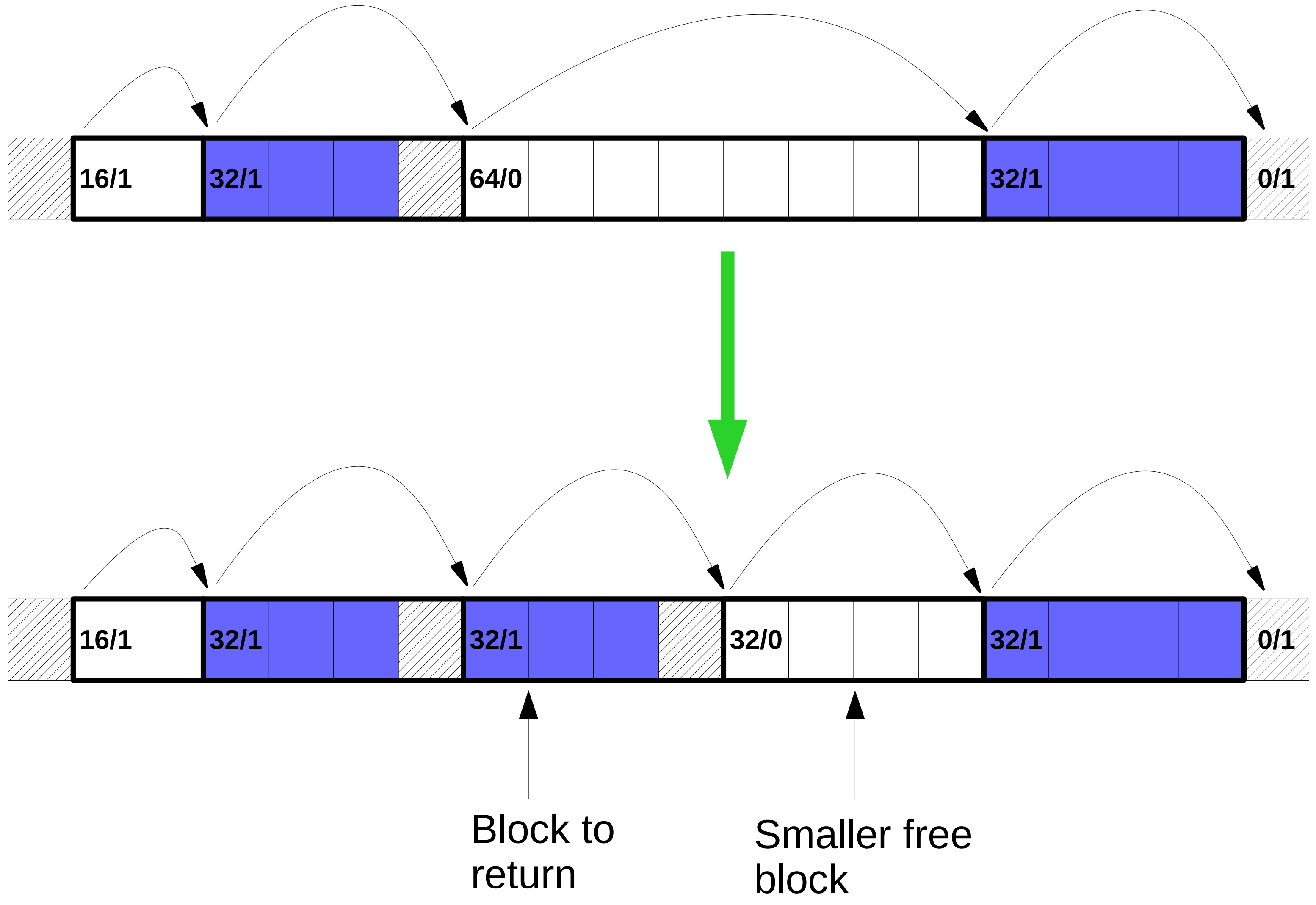
For example, given the request:
malloc(16)
We return the block that is 64 bytes long because it is free and large enough to satisfy the request:



Block size needed:
8 (header) + 16 (requested) + 8 (alignment) = 32;
actual size wastes 32 bytes (64 - 32)

What can we do about internal fragmentation?

Split the block!



Freeing a block

When the programmer passes the address of an allocated block to the free function, we must free the block.

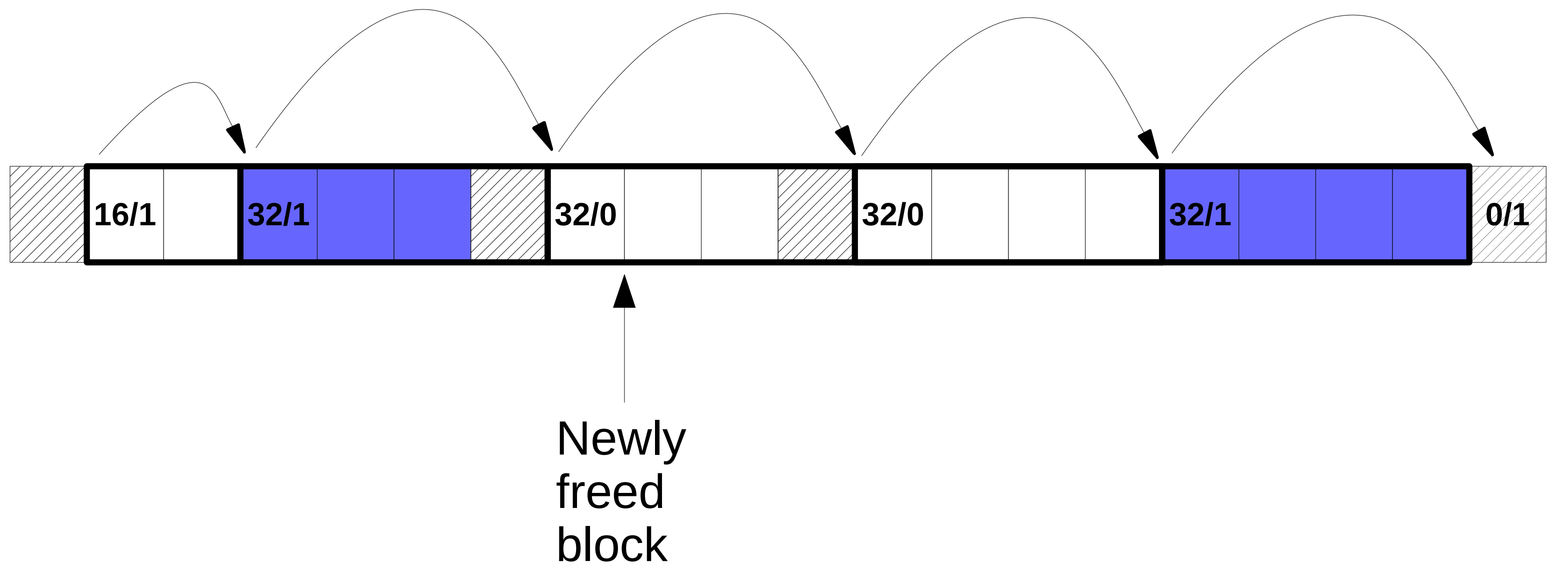
Technically, the only action we need to take in order to free a block is to change the allocation state bit from 1 to 0.

Note that the address that the programmer passes to free is the address of the payload, not the header. Thus, if the programmer passes the value held in the variable ptr, the header would begin one word earlier:

```
// find the beginning of the header
header_address= ptr – sizeof(char*)
// Note: header value = *header_address
// free the block
*header_address= (*header_address) & -2
```

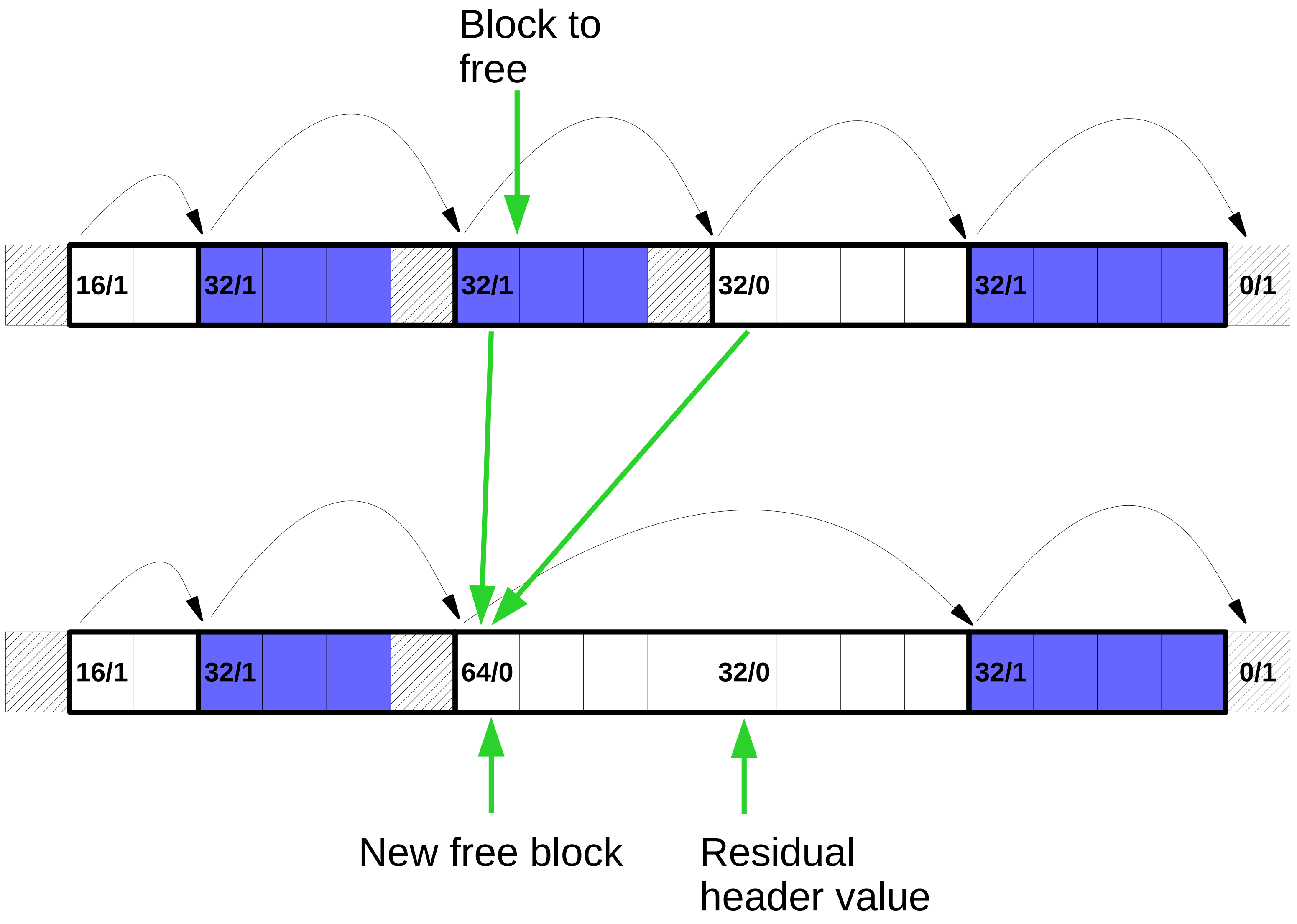
When we free a block, however, we can cause space to be wasted. This form of waste is known as external fragmentation.

Note that in the diagram below, there are 64 bytes available, but they are divided into two 32 byte blocks, thereby limiting the size we can allocate.



What can we do about this unnecessary external fragmentation?

We can coalesce adjacent free blocks!



Sample pseudocode to coalesce the next free block:

```
free( ptr )
    ptr= ptr - sizeof(char*)
    *ptr= *ptr & -2   // remote allocated bit
    next= ptr + *ptr   // get address of next block
    if ( (*next & 1) == 0 )   // if free
        *ptr= *ptr + *next   // combine sizes
```

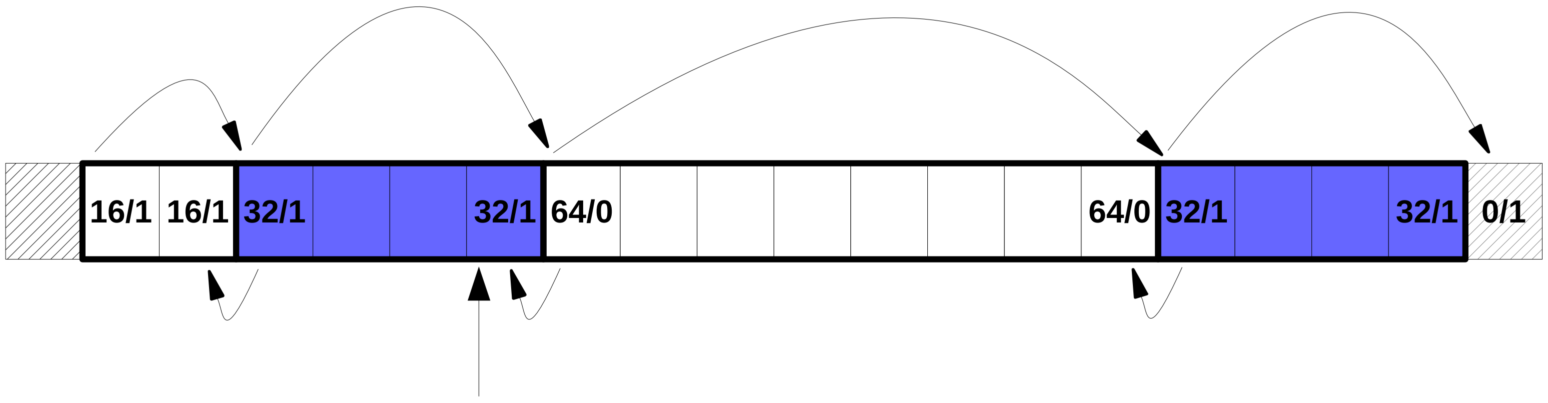
Note that the header data for the next block remains in place. As noted before, memory contains garbage before initialization; this includes blocks returned by malloc.

But what do we do if the block immediately preceding the block being freed is also free?

Answer: bidirectional coalescing

But how?

Knuth suggested “boundary tags,” i.e., footers that match the headers.

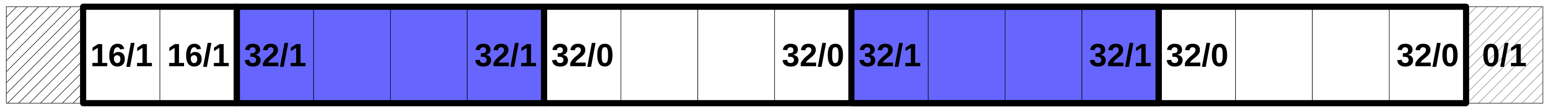


Footer starts at the address of the next block – word size

But what do we do when blocks are not adjacent?

We end up with gaps, which form another type of external fragmentation.

In the diagram below, there are 64 free bytes, but they are separated into non-adjacent blocks and, therefore, cannot be allocated as a single block.



Implicit list management performance

Freeing a block takes only constant time, even when coalescing preceding and following blocks.

Allocating a block takes linear time, which is considered poor performance.

Moreover, external fragmentation can be extensive and there is no available mitigation strategy.

Consequently, implicit list management is not used in production systems except for special purpose systems, such as small embedded systems.

Splitting and coalescing strategies, however, are used by explicit allocators, as well.

Summary of implicit list characteristics

Each block begins with a header that contains the size of the block in bytes and a bit that reflects whether the block is allocated or free.

Blocks are aligned on double-word boundaries.

Each block ends with a footer that matches the header.

The list is traversed in linear fashion to allocate blocks, using block size to find the next block.

Various strategies, such as first fit, next fit, best fit, etc., may be used to find a block to allocate.

When a block is allocated that is excessively large, it must be split into two blocks-- the block to be allocated and a new free block.

As blocks are freed, they are coalesced with adjacent free blocks.