CSE 4733/6733 - Operating System 1

Stephen A. Torri, Ph.D.

Mississippi State University

2023-02-13

CSE 4733/6733

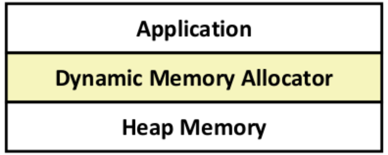
CSE 4733/6733 - Operating System 1

Stephen A. Torri, Ph.D.

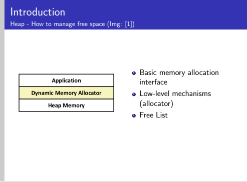
Mississippi State University

Introduction

Heap - How to manage free space (Img: [1])



- Basic memory allocation interface
- Low-level mechanisms (allocator)
- Free List


Introduction
Heap - How to manage free space (Img: [1])

Application
Dynamic Memory Allocator
Heap Memory

- Basic memory allocation interface
- Low-level mechanisms (allocator)
- Free List

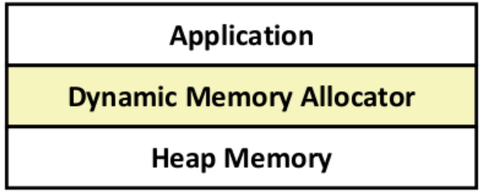
1. Focus of today's lecture is on how the dynamic memory is allocated, returned, and adjusted. A system call performs each action.

Memory API

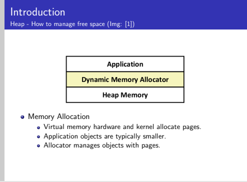
Memory API

Introduction

Heap - How to manage free space (Img: [1])

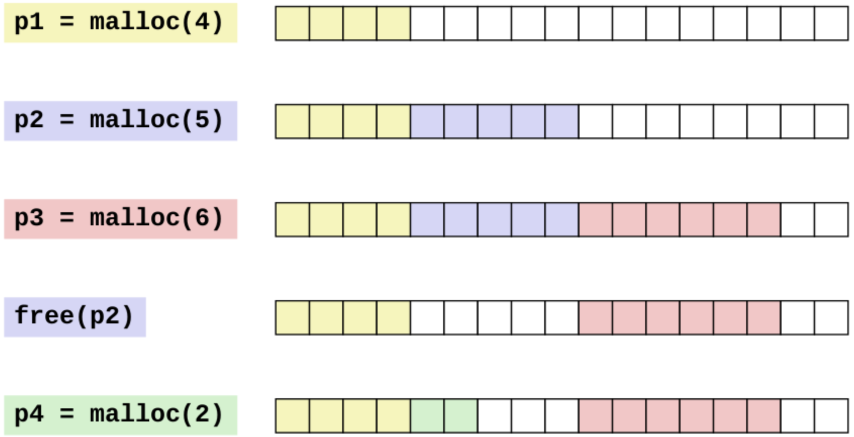


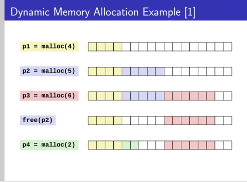
- Memory Allocation
  - Virtual memory hardware and kernel allocate pages.
  - Application objects are typically smaller.
  - Allocator manages objects with pages.


Introduction
Heap - How to manage free space (Img: [1])
Application
Dynamic Memory Allocator
Heap Memory
- Memory Allocation
  - Virtual memory hardware and kernel allocate pages.
  - Application objects are typically smaller.
  - Allocator manages objects with pages.

1. All the memory allocations are handled within the kernel space. No user application has direct access to system memory.

Dynamic Memory Allocation Example [1]

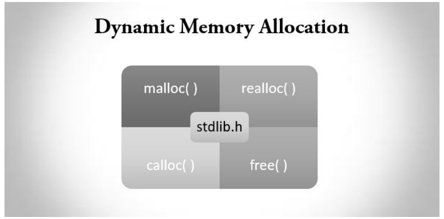




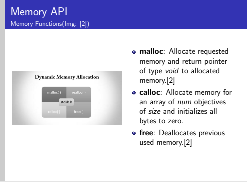
1. The operating system decides where to assign data in the heap memory.
2.

Memory API

Memory Functions(Img: [2])



- **malloc**: Allocate requested memory and return pointer of type *void* to allocated memory.[2]
- **calloc**: Allocate memory for an array of *num* objectives of *size* and initializes all bytes to zero.
- **free**: Deallocates previous used memory.[2]



1. Malloc: void *malloc(size_t size) - *size* is the number of bytes.
2. Free: void free(void* ptr) - *ptr* is a pointer that holds the address of the memory. This must be called for any previous call to malloc(), calloc(), or realloc()
3. Software developers need to ensure that any allocated memory is released after use.

Memory API

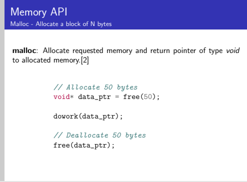
Malloc - Allocate a block of N bytes

malloc: Allocate requested memory and return pointer of type *void* to allocated memory.[2]

```
// Allocate 50 bytes
void* data_ptr = free(50);

dowork(data_ptr);

// Deallocate 50 bytes
free(data_ptr);
```


Memory API
Malloc - Allocate a block of N bytes

malloc: Allocate requested memory and return pointer of type *void* to allocated memory.[2]

```
// Allocate 50 bytes
void* data_ptr = free(50);

dowork(data_ptr);

// Deallocate 50 bytes
free(data_ptr);
```

1. Malloc() is a basic way to reserve memory. The size of the memory is the number of reserved bytes. So, in this case, 50 bytes are allocated.
2. Free() is the system call used to return the 50 bytes to the free list.

Memory API

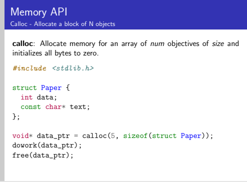
Calloc - Allocate a block of N objects

calloc: Allocate memory for an array of *num* objectives of *size* and initializes all bytes to zero.

```c
#include <stdlib.h>

struct Paper {
  int data;
  const char* text;
};

void* data_ptr = calloc(5, sizeof(struct Paper));
dowork(data_ptr);
free(data_ptr);
```


Memory API
Calloc - Allocate a block of N objects

calloc: Allocate memory for an array of *num* objectives of *size* and initializes all bytes to zero.

```c
#include <stdlib.h>

struct Paper {
  int data;
  const char* text;
};

void* data_ptr = calloc(5, sizeof(struct Paper));
dowork(data_ptr);
free(data_ptr);
```

1. Calloc is useful to allocate memory for a type. In this example, the size of "struct" is about 8 bytes (32 bits). Instead of declaring the bytes, it is easier to use the sizeof() function to calculate the required memory. Now any changes to Paper will not require changing the memory allocation.

Memory API

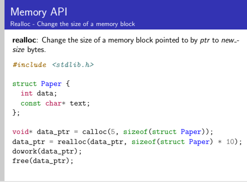
Realloc - Change the size of a memory block

realloc: Change the size of a memory block pointed to by *ptr* to *new_-size* bytes.

```
#include <stdlib.h>

struct Paper {
  int data;
  const char* text;
};

void* data_ptr = calloc(5, sizeof(struct Paper));
data_ptr = realloc(data_ptr, sizeof(struct Paper) * 10);
dowork(data_ptr);
free(data_ptr);
```


Memory API
Realloc - Change the size of a memory block

realloc: Change the size of a memory block pointed to by *ptr* to *new_-size* bytes.

```
#include <stdlib.h>

struct Paper {
  int data;
  const char* text;
};

void* data_ptr = calloc(5, sizeof(struct Paper));
data_ptr = realloc(data_ptr, sizeof(struct Paper) * 10);
dowork(data_ptr);
free(data_ptr);
```

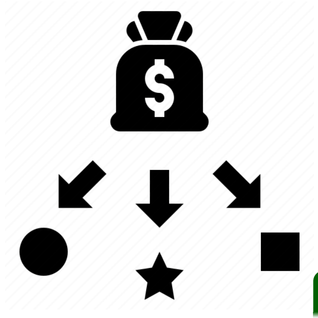
1. Requirements:
   - Pointer is not NULL
   - Memory previously alocated by *malloc()*, *calloc*, or *realloc*
   - Memory is not deallocated by a call to *calloc* or resized by a call to *realloc*.
2. Reallocation can be done by:
   - Expand or contract the existing area pointer to by *ptr*.
   - Move:
     - Allocate a new block of memory to *new_size*
     - Copy memory bytes from old memory to new memory location.
     - Free the old memory.

Low-Level Mechanisms

Low-Level Mechanisms

Allocator Duty[3]

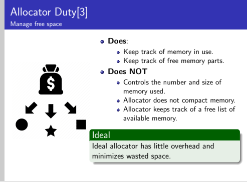
Manage free space



- **Does**:
  - Keep track of memory in use.
  - Keep track of free memory parts.
- **Does NOT**
  - Controls the number and size of memory used.
  - Allocator does not compact memory.
  - Allocator keeps track of a free list of available memory.

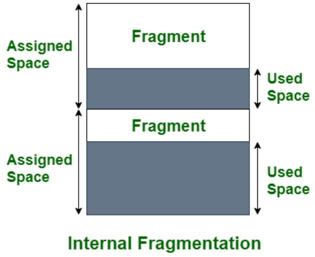
Ideal

Ideal allocator has little overhead and minimizes wasted space.



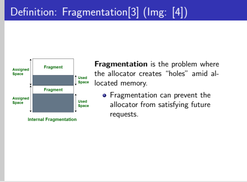
1. An allocator is, therefore, an online algorithm that must respond to requests in strict sequence immediately, and its decisions are irrevocable[3]

Definition: Fragmentation[3] (Img: [4])



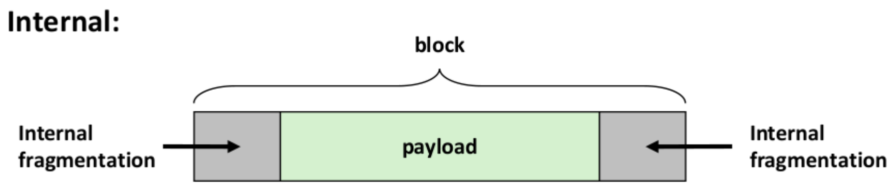
Fragmentation is the problem where the allocator creates "holes" amid allocated memory.

- Fragmentation can prevent the allocator from satisfying future requests.

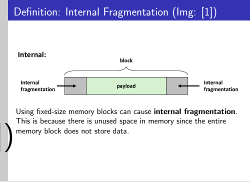


1. The repeated calls to malloc() and free() will cause the system memory to be eventually reduced to small segments of memory. To allocate large memory will necessitate the compaction of system memory.

Definition: Internal Fragmentation (Img: [1])



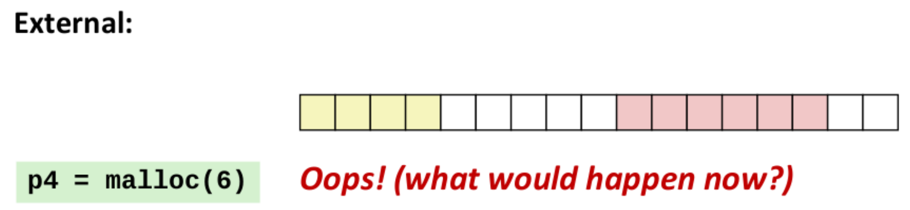
Using fixed-size memory blocks can cause **internal fragmentation**. This is because there is unused space in memory since the entire memory block does not store data.



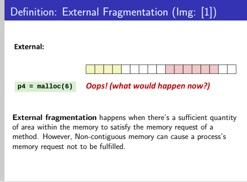
1. – Caused by[1]
   - Overhead of maintaining heap data structures
   - Padding for alignment purposes
   - Explicit policy decisions (e.g., to return a big block to satisfy a small request)

   – Depends only on the pattern of previous requests.
   - Thus, easy to measure.

Definition: External Fragmentation (Img: [1])



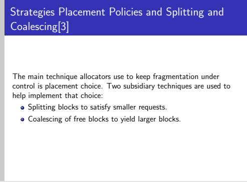
External fragmentation happens when there's a sufficient quantity of area within the memory to satisfy the memory request of a method. However, Non-contiguous memory can cause a process's memory request not to be fulfilled.



Strategies Placement Policies and Splitting and Coalescing[3]

The main technique allocators use to keep fragmentation under control is placement choice. Two subsidiary techniques are used to help implement that choice:

- Splitting blocks to satisfy smaller requests.
- Coalescing of free blocks to yield larger blocks.



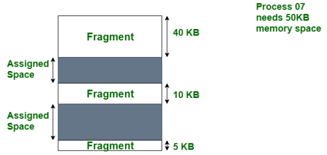
Strategies Placement Policies and Splitting and Coalescing[3]

The main technique allocators use to keep fragmentation under control is placement choice. Two subsidiary techniques are used to help implement that choice:

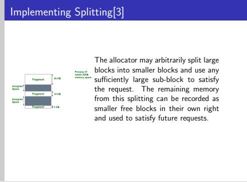
- Splitting blocks to satisfy smaller requests.
- Coalescing of free blocks to yield larger blocks.

1. The allocator can place a requested block anywhere it can find a sufficiently large range of free memory with enough space.[3]
2. An allocator algorithm, therefore, should be regarded as the mechanism that implements a placement policy that is motivated by a strategy for minimizing fragmentation.[3]

Implementing Splitting[3]



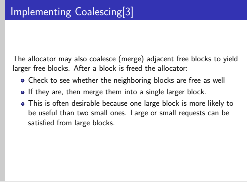
The allocator may arbitrarily split large blocks into smaller blocks and use any sufficiently large sub-block to satisfy the request. The remaining memory from this splitting can be recorded as smaller free blocks in their own right and used to satisfy future requests.



Implementing Coalescing[3]

The allocator may also coalesce (merge) adjacent free blocks to yield larger free blocks. After a block is freed the allocator:

- Check to see whether the neighboring blocks are free as well
- If they are, then merge them into a single larger block.
- This is often desirable because one large block is more likely to be useful than two small ones. Large or small requests can be satisfied from large blocks.


Implementing Coalescing[3]

The allocator may also coalesce (merge) adjacent free blocks to yield larger free blocks. After a block is freed the allocator:

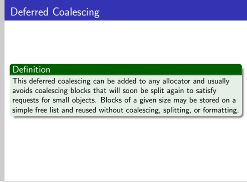
- Check to see whether the neighboring blocks are free as well
- If they are, then merge them into a single larger block.
- This is often desirable because one large block is more likely to be useful than two small ones. Large or small requests can be satisfied from large blocks.

1. Memory access coalescing is an optimization to coalesce narrow memory references with contiguous addresses into a wider reference for taking advantage of a wide-bus architecture.[5]
2. Coalescing memory accesses can improve performance for two reasons: one by reducing the additional cycles required for moving data from caches to registers and the other by reducing the stall cycles caused by multiple outstanding memory access requests.[5]

Deferred Coalescing

Definition

This deferred coalescing can be added to any allocator and usually avoids coalescing blocks that will soon be split again to satisfy requests for small objects. Blocks of a given size may be stored on a simple free list and reused without coalescing, splitting, or formatting.



Deferred Coalescing

Definition

This deferred coalescing can be added to any allocator and usually avoids coalescing blocks that will soon be split again to satisfy requests for small objects. Blocks of a given size may be stored on a simple free list and reused without coalescing, splitting, or formatting.

1. While deferred coalescing is traditionally thought of as speed optimization, it is important to note that fragmentation considerations come into play in three ways:[3]
   – The lower fragmentation is, the more important deferred coalescing will be in speed.
   – Deferred coalescing may significantly affect fragmentation by changing the allocator's decisions regarding which memory blocks to hold which objects.
   – Deferred coalescing may decrease the locality of reference for the same reason because recently freed small blocks will usually not be reused to hold larger objects.

Keeping Track of Free Blocks

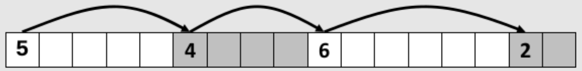
Keeping Track of Free Blocks

Free Block Management Algorithms[1]

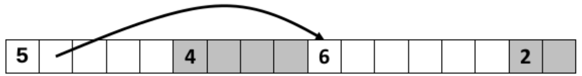
Method 1: Implicit List

Keeping Track of Free Blocks

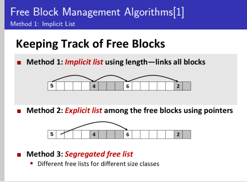
- **Method 1:** ***Implicit list*** **using length—links all blocks**



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



- **Method 3:** ***Segregated free list***
  - Different free lists for different size classes

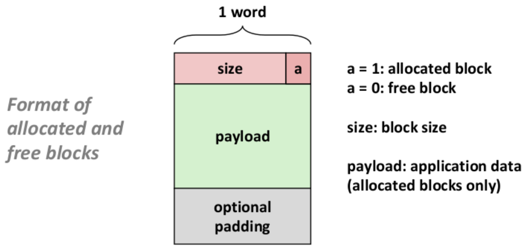

Free Block Management Algorithms[1]
Method 1: Implicit List
Keeping Track of Free Blocks
Method 1: Implicit list using length—links all blocks
Method 2: Explicit list among the free blocks using pointers
Method 3: Segregated free list
Different free lists for different size classes

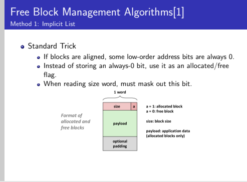
1. In the implicit list, all the use and free blocks are linked together.

Free Block Management Algorithms[1]

Method 1: Implicit List

- Standard Trick
  - If blocks are aligned, some low-order address bits are always 0.
  - Instead of storing an always-0 bit, use it as an allocated/free flag.
  - When reading size word, must mask out this bit.



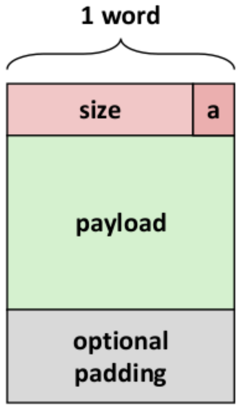


1. To allocate new memory, the allocator has to walk the entire memory list to find a segment that is sufficiently big enough to fulfill the request.
2. Worse case, the allocator walks the entire memory space searching for an adequately sized memory segment.

Free Block Management Algorithms[1]

Implicit List

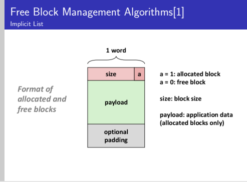
Format of allocated and free blocks



a = 1: allocated block
a = 0: free block

size: block size

payload: application data (allocated blocks only)

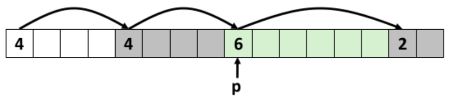

Free Block Management Algorithms[1]
Implicit List
1 word
size a
payload
optional padding
Format of allocated and free blocks
a = 1: allocated block
a = 0: free block
size: block size
payload: application data (allocated blocks only)

Free Block Management Algorithms[1]

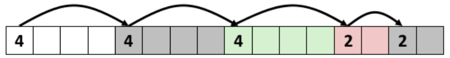
Implicit List - Splitting

Implicit List: Allocating in Free Block

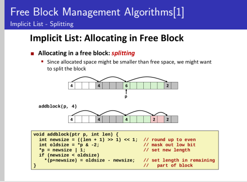
- **Allocating in a free block:** ***splitting***
  - Since allocated space might be smaller than free space, we might want to split the block



addblock(p, 4)



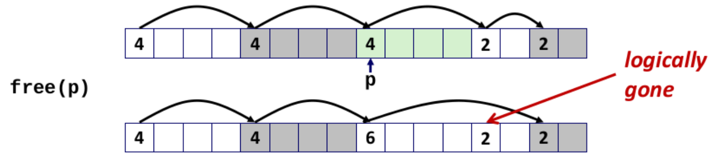
```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;  // round up to even
  int oldsize = *p & -2;                // mask out low bit
  *p = newsize | 1;                     // set new length
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;   // set length in remaining
}                                       //   part of block
```



1. Splitting memory adds a new entry to the free memory list. As the number of small segments increases, the longer it will take to walk the entire memory location.

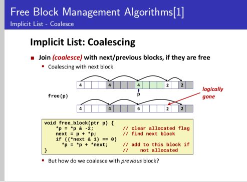
Implicit List: Coalescing

- **Join *(coalesce)* with next/previous blocks, if they are free**
  - Coalescing with next block



```
void free_block(ptr p) {
    *p = *p & -2;          // clear allocated flag
    next = p + *p;         // find next block
    if ((*next & 1) == 0)
      *p = *p + *next;     // add to this block if
}                          //    not allocated
```

  - But how do we coalesce with *previous* block?



1. Coalescing of memory is necessary when there is no memory location big enough to satisfy a request.

Free Block Management Algorithms[1]

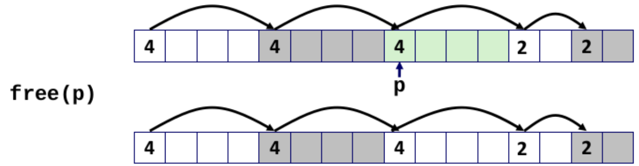
Implicit List - Free

Implicit List: Freeing a Block

- **Simplest implementation:**
  - Need only clear the "allocated" flag

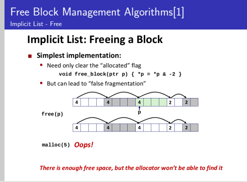
    ```
    void free_block(ptr p) { *p = *p & -2 }
    ```

  - But can lead to "false fragmentation"



`malloc(5)` ***Oops!***

There is enough free space, but the allocator won't be able to find it

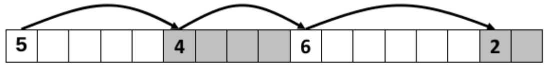


Free Block Management Algorithms[1]

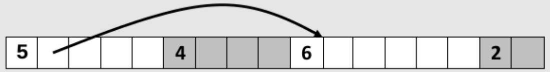
Method 2: Explicit List

Keeping Track of Free Blocks

- **Method 1:** ***Implicit free list*** **using length—links all blocks**



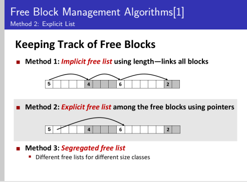
- **Method 2:** ***Explicit free list*** **among the free blocks using pointers**



- **Method 3:** ***Segregated free list***
  - Different free lists for different size classes

2023-02-13

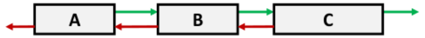
CSE 4733/6733
└Keeping Track of Free Blocks
└Free Block Management Algorithms[1]


Free Block Management Algorithms[1]
Method 2: Explicit List
Keeping Track of Free Blocks
Method 1: Implicit free list using length—links all blocks
Method 2: Explicit free list among the free blocks using pointers
Method 3: Segregated free list
Different free lists for different size classes

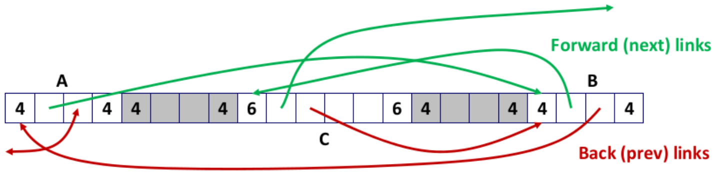
Free Block Management Algorithms[1]

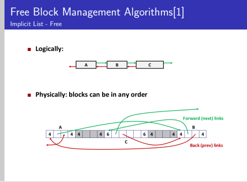
Implicit List - Free

- **Logically:**



- **Physically: blocks can be in any order**



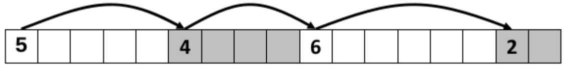


Free Block Management Algorithms[1]

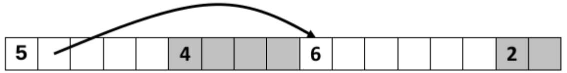
Method 3: Segregated Free List

Keeping Track of Free Blocks

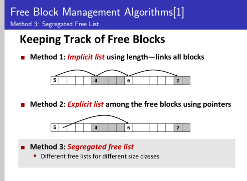
- **Method 1: *Implicit list* using length—links all blocks**



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



- **Method 3: *Segregated free list***
  - Different free lists for different size classes

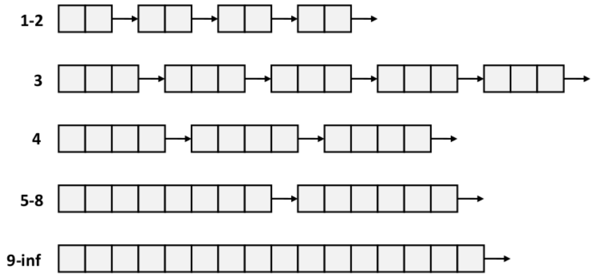


Free Block Management Algorithms[1]

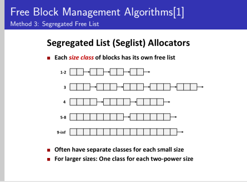
Method 3: Segregated Free List

Segregated List (Seglist) Allocators

- Each *size class* of blocks has its own free list



- Often have separate classes for each small size
- For larger sizes: One class for each two-power size



References I

Roger Dannenberg and Greg Ganger, 2009.

Unknown.
Dynamic memory allocation in c programming, 2023.

Paul R Wilson, Mark S Johnstone, Michael Neely, and David Boles.
Dynamic storage allocation: A survey and critical review.
In *Memory Management: International Workshop IWMM 95 Kinross, UK, September 27–29, 1995 Proceedings*, pages 1–116. Springer, 1995.

Unknown, 2022.

Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani.
Instruction combining for coalescing memory accesses using global code motion.
In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance, MSP 2004*, pages 2–11, 01 2004.

References I

Roger Dannenberg and Greg Ganger, 2009.

Unknown.
Dynamic memory allocation in c programming, 2023.

Paul R Wilson, Mark S Johnstone, Michael Neely, and David Boles.
Dynamic storage allocation: A survey and critical review.
In *Memory Management: International Workshop IWMM 95 Kinross, UK, September 27–29, 1995 Proceedings*, pages 1–116. Springer, 1995.

Unknown, 2022.

Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani.
Instruction combining for coalescing memory accesses using global code motion.
In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance, MSP 2004*, pages 2–11, 01 2004.