



# Dynamic Memory Management



# Goals of this Lecture

## Help you learn about:

- The need for dynamic\* memory mgmt (DMM)
- Implementing DMM using the heap section
- Implementing DMM using virtual memory

\* During program execution

# System-Level Functions Covered



As noted in the *Exceptions and Processes* lecture...

Linux system-level functions for **dynamic memory management (DMM)**

Number	Function	Description
45	brk()	Move the program break, thus changing the amount of memory allocated to the HEAP
45	sbrk()	(Variant of previous)
90	mmap()	Map a virtual memory page
91	munmap()	Unmap a virtual memory page

# Goals for DMM



## Goals for effective DMM:

- **Time** efficiency
  - Allocating and freeing memory should be fast
- **Space** efficiency
  - Pgm should use little memory

## Note

- Easy to reduce time **or** space
- Hard to reduce time **and** space

# Agenda



## The need for DMM

DMM using the heap section

DMMgr 1: Minimal implementation

DMMgr 2: Pad implementation

Fragmentation

DMMgr 3: List implementation

DMMgr 4: Doubly-linked list implementation

DMMgr 5: Bins implementation

DMM using virtual memory

DMMgr 6: VM implementation

# Why Allocate Memory Dynamically?



## Why **allocate** memory dynamically?

### Problem

- Unknown object size
  - E.g. unknown element count in array
  - E.g. unknown node count in linked list or tree
- How much memory to allocate?

### Solution 1

- Guess!

### Solution 2

- Allocate memory dynamically

# Why Free Memory Dynamically?



## Why **free** memory dynamically?

### Problem

- Pgm should use little memory, i.e.
- Pgm should **map** few pages of virtual memory
  - Mapping unnecessary VM pages bloats page tables, wastes memory/disk space

### Solution

- Free dynamically allocated memory that is no longer needed



# Option 1: Automatic Freeing

## Run-time system frees unneeded memory

- Java, Python, ...
- **Garbage collection**

### Pros:

- Easy for programmer

### Cons:

- Performed constantly => overhead
- Performed periodically => unexpected pauses

```
Car c;  
Plane p;  
...  
c = new Car();  
p = new Plane();  
...  
c = new Car();  
...
```

Original Car  
object can't  
be accessed





# Option 2: Manual Freeing

## Programmer frees unneeded memory

- C, C++, Objective-C, ...

## Pros

- No overhead
- No unexpected pauses

## Cons

- More complex for programmer
- Opens possibility of memory-related bugs
  - Dereferences of dangling pointers, double frees, memory leaks

We'll focus on **manual** freeing

# Standard C DMM Functions



Standard C DMM functions:

```
void *malloc(size_t size);  
void free(void *ptr);  
void *calloc(size_t nmem, size_t size);  
void *realloc(void *ptr, size_t size);
```

Collectively define a **dynamic memory manager (DMMgr)**

We'll focus on `malloc()` and `free()`

# Implementing malloc() and free()



## Question:

- How to implement `malloc()` and `free()`?
- How to implement a DMMgr?

## Answer 1:

- Use the heap section of memory

## Answer 2:

- (Later in this lecture)

# Agenda



The need for DMM

## **DMM using the heap section**

DMMgr 1: Minimal implementation

DMMgr 2: Pad implementation

Fragmentation

DMMgr 3: List implementation

DMMgr 4: Doubly-linked list implementation

DMMgr 5: Bins implementation

DMM using virtual memory

DMMgr 6: VM implementation

# The Heap Section of Memory



Supported by Unix/Linux, MS Windows, ...

Heap start is stable

**Program break** points to end

At process start-up, heap start == program break

Can grow dynamically

By moving program break to higher address

Thereby (indirectly) mapping pages of virtual mem

Can shrink dynamically

By moving program break to lower address

Thereby (indirectly) unmapping pages of virtual mem



# Unix Heap Management

Unix system-level functions for heap mgmt:

```
int brk(void *p) ;
```

- Move the program break to address **p**
- Return 0 if successful and -1 otherwise

```
void *sbrk(intptr_t n) ;
```

- Increment the program break by **n** bytes
- **If n is 0, then return the current location of the program break**
- Return 0 if successful and (void\*)-1 otherwise
- **Beware: On Linux has a known bug (overflow not handled); should call only with argument 0.**

Note: minimal interface (good!)

# Agenda



The need for DMM

DMM using the heap section

**DMMgr 1: Minimal implementation**

DMMgr 2: Pad implementation

Fragmentation

DMMgr 3: List implementation

DMMgr 4: Doubly-linked list implementation

DMMgr 5: Bins implementation

DMM using virtual memory

DMMgr 6: VM implementation

# Minimal Impl



## Data structures

- None!

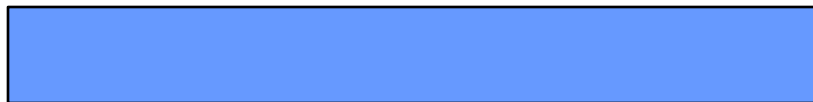
## Algorithms (by examples)...



# Minimal Impl malloc(n) Example

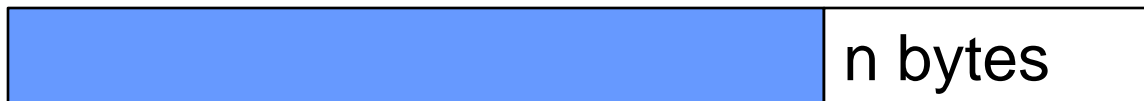


Call `sbrk(0)` to determine current program break (`p`)



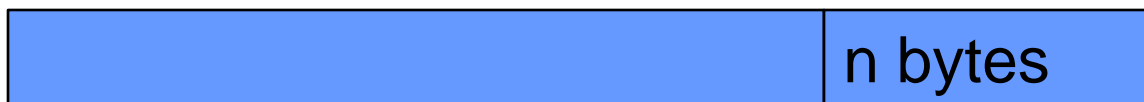
↑  
`p`

Call `brk(p+n)` to increase heap size



↑  
`p`

Return `p`



↑  
`p`

# Minimal Impl free(p) Example



Do nothing!



# Minimal Impl



## Algorithms

```
void *malloc(size_t n)
{  char *p = sbrk(0);
   if (brk(p + n) == -1)
       return NULL;
   return p;
}
```

```
void free(void *p)
{
}
```

# Minimal Impl Performance



## Performance (general case)

- **Time:** bad
  - Two system calls per `malloc()`
- **Space:** bad
  - Each call of `malloc()` extends heap size
  - No reuse of freed chunks

# What's Wrong?



## Problem

- `malloc()` executes two system calls

## Solution

- Redesign `malloc()` so it does fewer system calls
- Maintain a pad at the end of the heap...

# Agenda



The need for DMM

DMM using the heap section

DMMgr 1: Minimal implementation

**DMMgr 2: Pad implementation**

Fragmentation

DMMgr 3: List implementation

DMMgr 4: Doubly-linked list implementation

DMMgr 5: Bins implementation

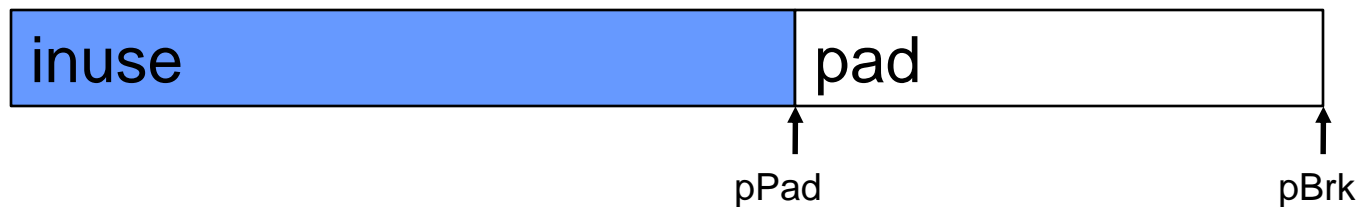
DMM using virtual memory

DMMgr 6: VM implementation



# Pad Impl

## Data structures



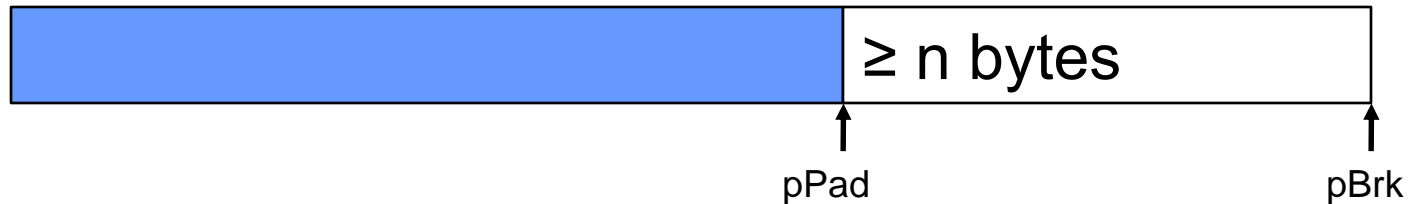
- **pBrk**: address of end of heap (i.e. the program break)
- **pPad**: address of beginning of pad

```
char *pPad = NULL;  
char *pBrk = NULL;
```

Algorithms (by examples)...



# Pad Impl malloc(n) Example 1



Are there at least  $n$  bytes between  $pPad$  and  $pBrk$ ? **Yes!**  
Save  $pPad$  as  $p$ ; add  $n$  to  $pPad$



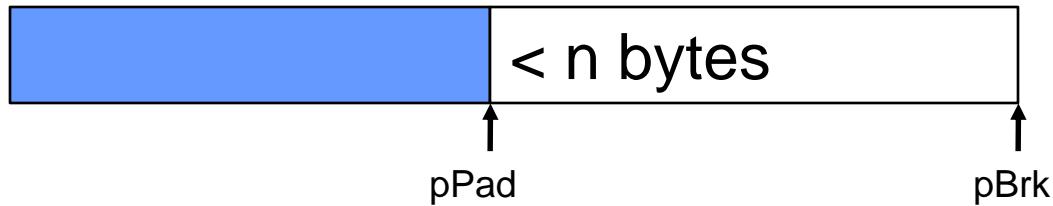
Return  $p$



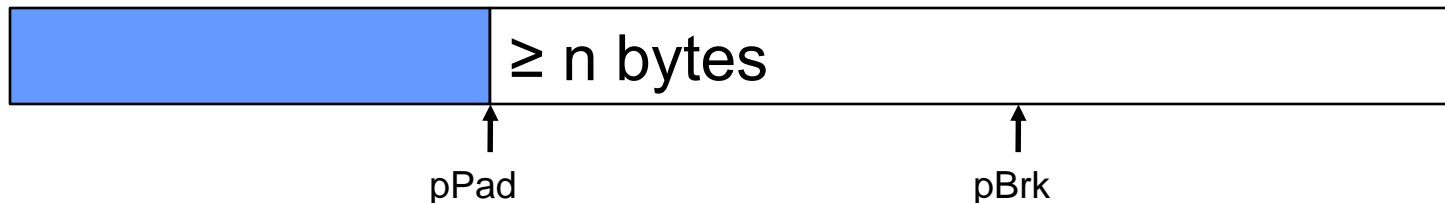




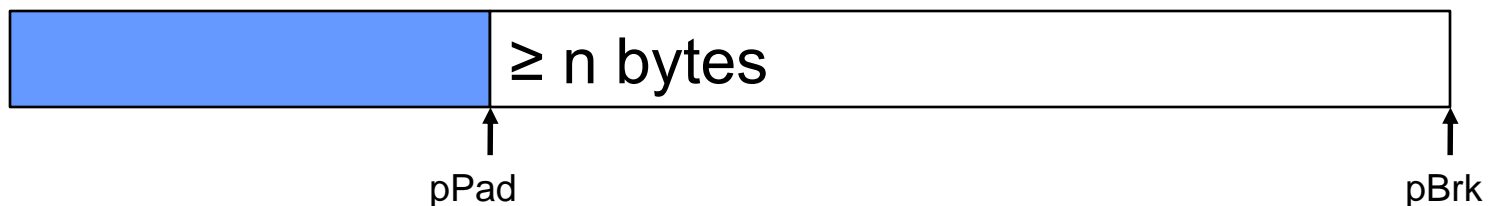
# Pad Impl malloc(n) Example 2



Are there at least  $n$  bytes between `pPad` and `pBrk`? **No!**  
Call `brk()` to allocate (more than) enough additional memory



Set `pBrk` to new program break



Proceed as previously!

# Pad Impl free(p) Example



Do nothing!





# Pad Impl



## Algorithms

```
void *malloc(size_t n)
{ enum {MIN_ALLOC = 4096};
  char *p;
  char *pNewBrk;
  if (pBrk == NULL)
  { pBrk = sbrk(0);
    pPad = pBrk;
  }
```

```
void free(void *p)
{
}
```

```
if (pPad + n > pBrk) /* move pBrk */
{ pNewBrk =
  max(pPad + n, pBrk + MIN_ALLOC);
  if (brk(pNewBrk) == -1) return NULL;
  pBrk = pNewBrk;
}
p = pPad;
pPad += n;
return p;
}
```

# Pad Impl Performance



## Performance (general case)

- **Time:** good
  - `malloc()` calls `sbrk()` initially
  - `malloc()` calls `brk()` infrequently thereafter
- **Space:** bad
  - No reuse of freed chunks

# What's Wrong?



## Problem

- `malloc()` doesn't reuse freed chunks

## Solution

- `free()` marks freed chunks as "free"
- `malloc()` uses marked chunks whenever possible
- `malloc()` extends size of heap only when necessary

# Agenda



The need for DMM

DMM using the heap section

DMMgr 1: Minimal implementation

DMMgr 2: Pad implementation

## **Fragmentation**

DMMgr 3: List implementation

DMMgr 4: Doubly-linked list implementation

DMMgr 5: Bins implementation

DMM using virtual memory

DMMgr 6: VM implementation

# Fragmentation



At any given time, some heap memory chunks are in use, some are marked “free”



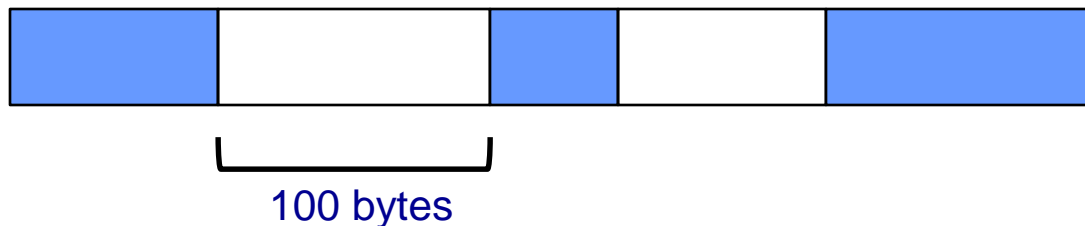
DMMgr must be concerned about **fragmentation...**



# Internal Fragmentation

**Internal fragmentation:** waste **within** chunks

Example



Client asks for 90 bytes

DMMgr provides chunk of size 100 bytes

10 bytes wasted

Generally

Program asks for  $n$  bytes

DMMgr provides chunk of size  $n + \Delta$  bytes

$\Delta$  bytes wasted

Space efficiency =>

DMMgr should reduce internal fragmentation

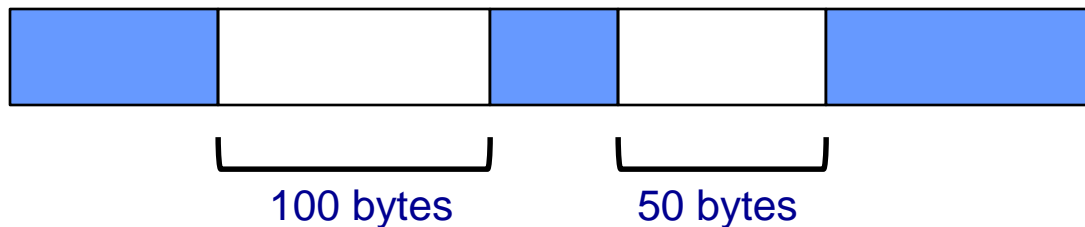




# External Fragmentation

**External fragmentation:** waste **between** chunks

Example



Client asks for 150 bytes

150 bytes are available, but not contiguously

DMMgr must extend size of heap

Generally

Program asks for  $n$  bytes

$n$  bytes are available, but not contiguously

DMMgr must extend size of heap to satisfy request

Space efficiency =>

DMMgr should reduce external fragmentation

# DMMgr Desired Behavior Demo

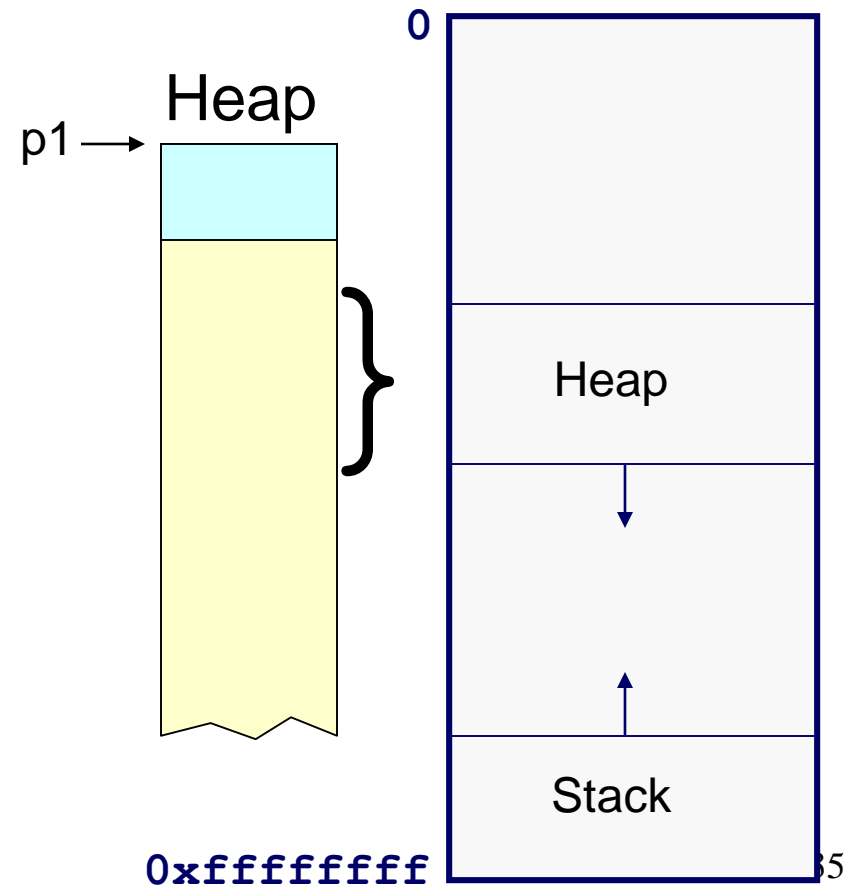


```
char *p1 = malloc(3);  
char *p2 = malloc(1);  
char *p3 = malloc(4);  
free(p2);  
char *p4 = malloc(6);  
free(p3);  
char *p5 = malloc(2);  
free(p1);  
free(p4);  
free(p5);
```

# DMMgr Desired Behavior Demo



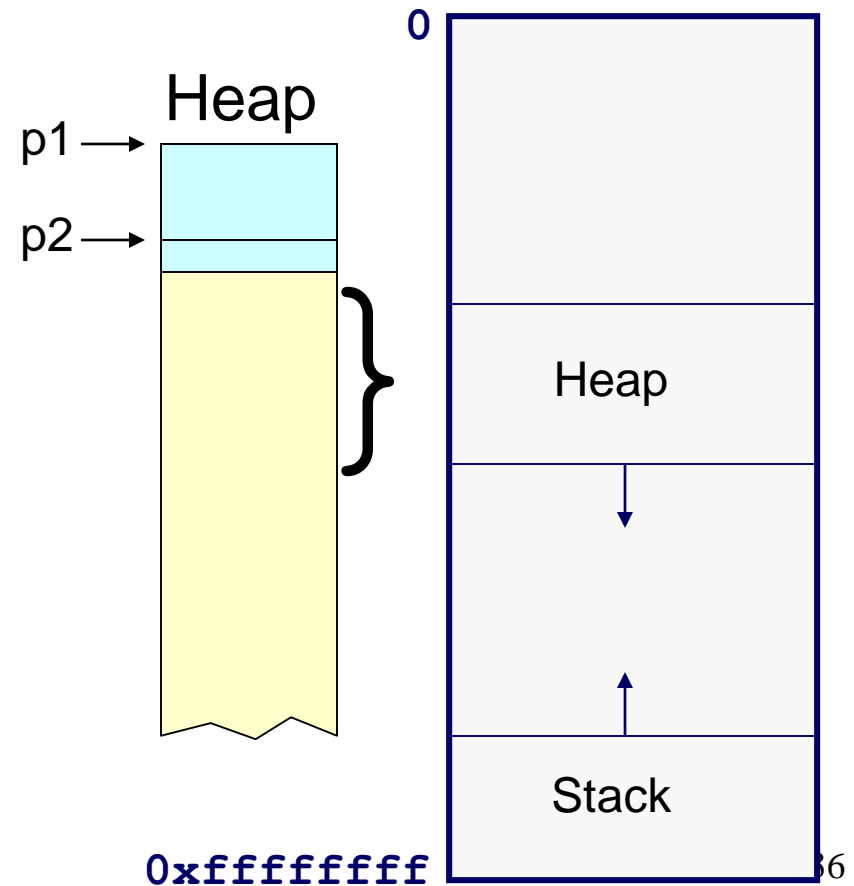
```
➔ char *p1 = malloc(3);  
char *p2 = malloc(1);  
char *p3 = malloc(4);  
free(p2);  
char *p4 = malloc(6);  
free(p3);  
char *p5 = malloc(2);  
free(p1);  
free(p4);  
free(p5);
```



# DMMgr Desired Behavior Demo



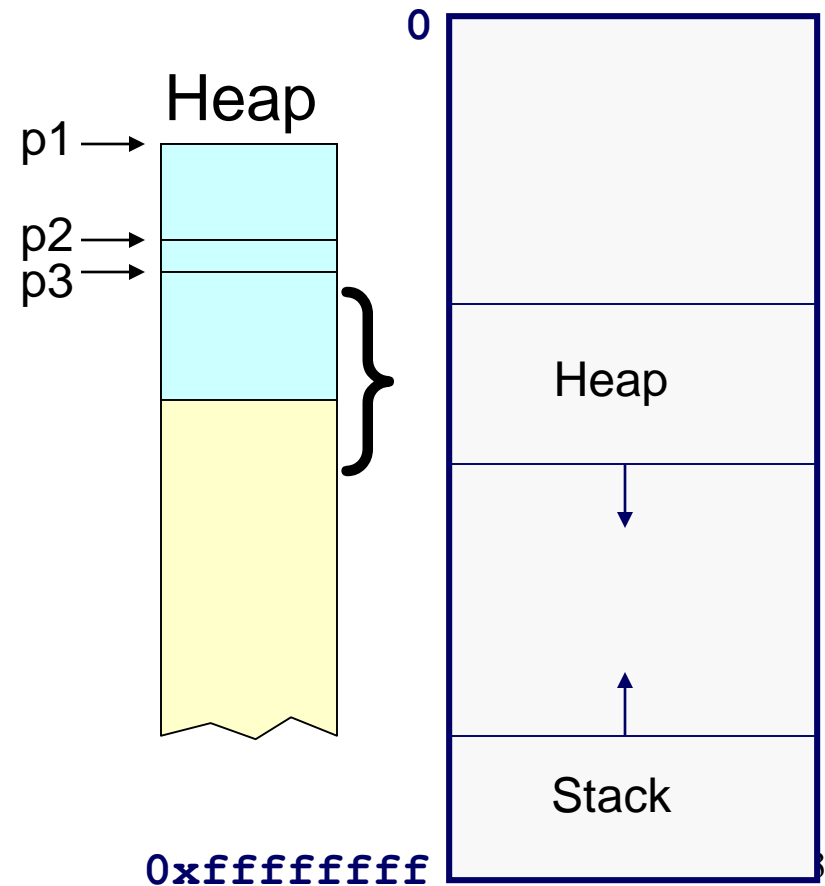
```
→ char *p1 = malloc(3);  
char *p2 = malloc(1);  
char *p3 = malloc(4);  
free(p2);  
char *p4 = malloc(6);  
free(p3);  
char *p5 = malloc(2);  
free(p1);  
free(p4);  
free(p5);
```



# DMMgr Desired Behavior Demo



```
char *p1 = malloc(3);  
char *p2 = malloc(1);  
char *p3 = malloc(4);  
free(p2);  
char *p4 = malloc(6);  
free(p3);  
char *p5 = malloc(2);  
free(p1);  
free(p4);  
free(p5);
```



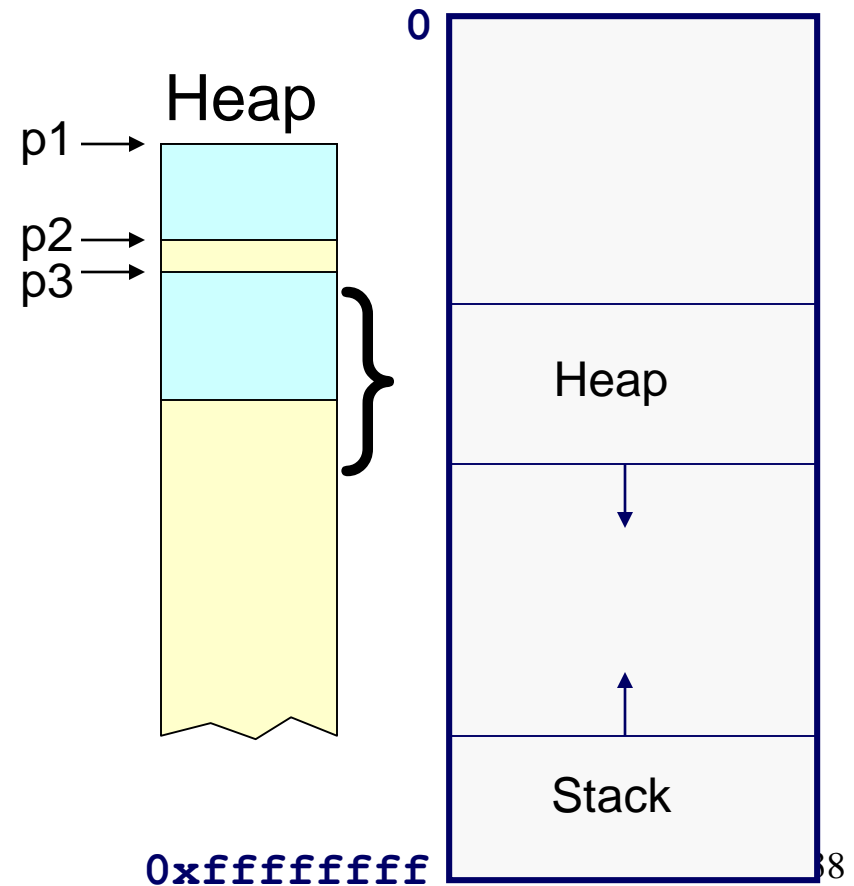
0xffffffff

# DMMgr Desired Behavior Demo



External fragmentation occurred

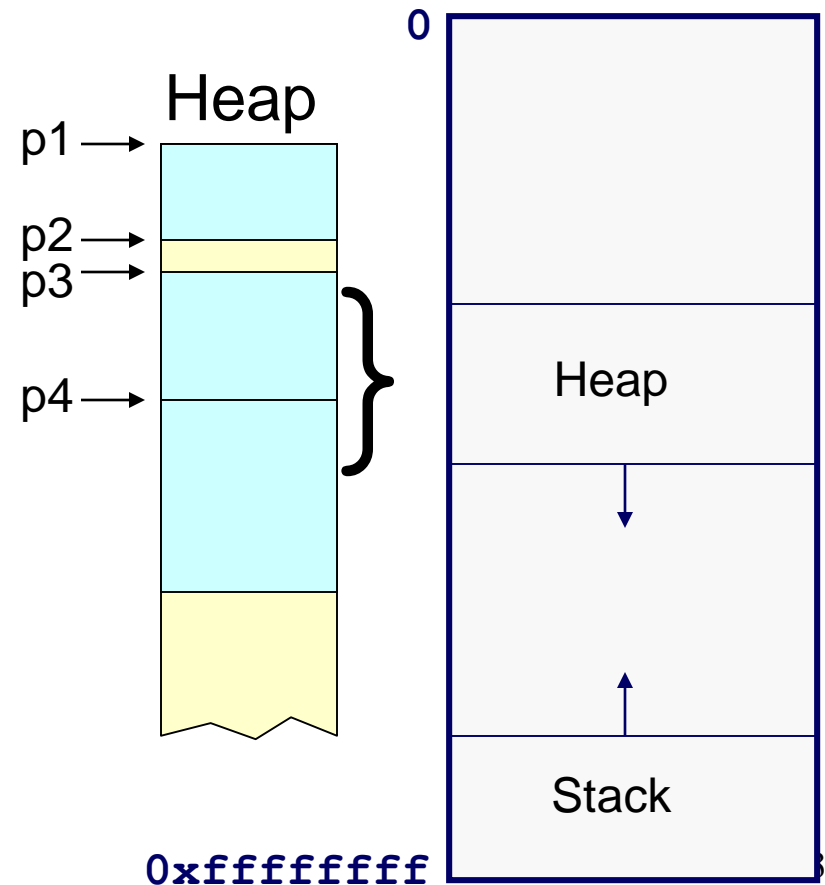
```
char *p1 = malloc(3);  
char *p2 = malloc(1);  
char *p3 = malloc(4);  
→ free(p2);  
char *p4 = malloc(6);  
free(p3);  
char *p5 = malloc(2);  
free(p1);  
free(p4);  
free(p5);
```



# DMMgr Desired Behavior Demo



```
char *p1 = malloc(3);  
char *p2 = malloc(1);  
char *p3 = malloc(4);  
free(p2);  
→ char *p4 = malloc(6);  
free(p3);  
char *p5 = malloc(2);  
free(p1);  
free(p4);  
free(p5);
```

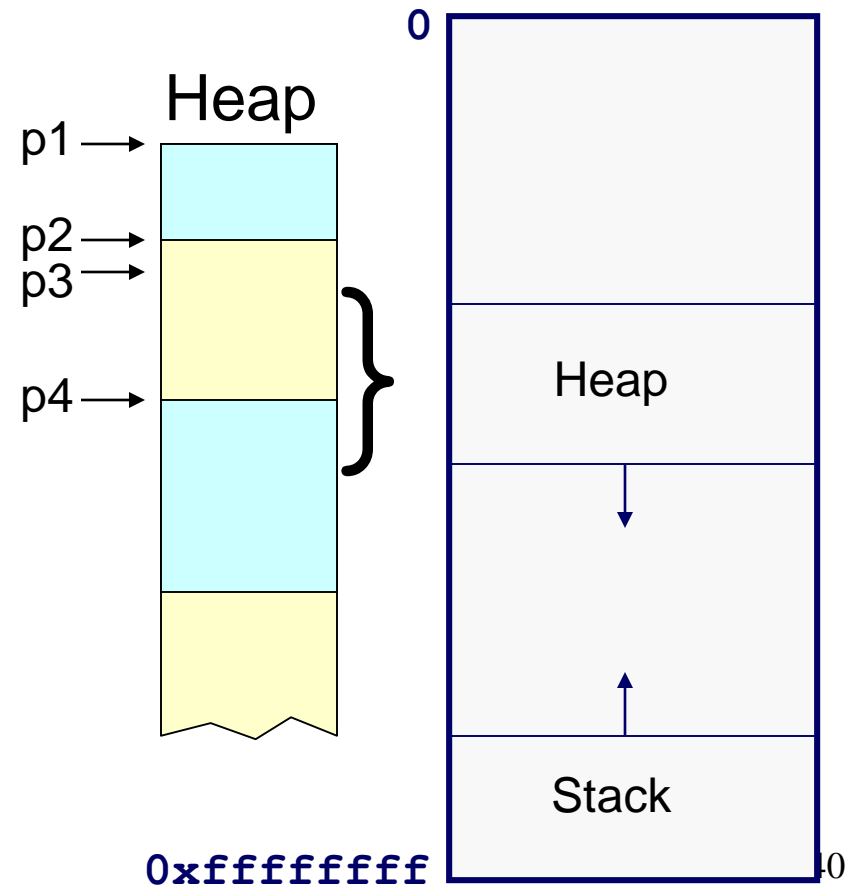


# DMMgr Desired Behavior Demo



DMMgr coalesced two free chunks

```
char *p1 = malloc(3);  
char *p2 = malloc(1);  
char *p3 = malloc(4);  
free(p2);  
char *p4 = malloc(6);  
free(p3);  
char *p5 = malloc(2);  
free(p1);  
free(p4);  
free(p5);
```



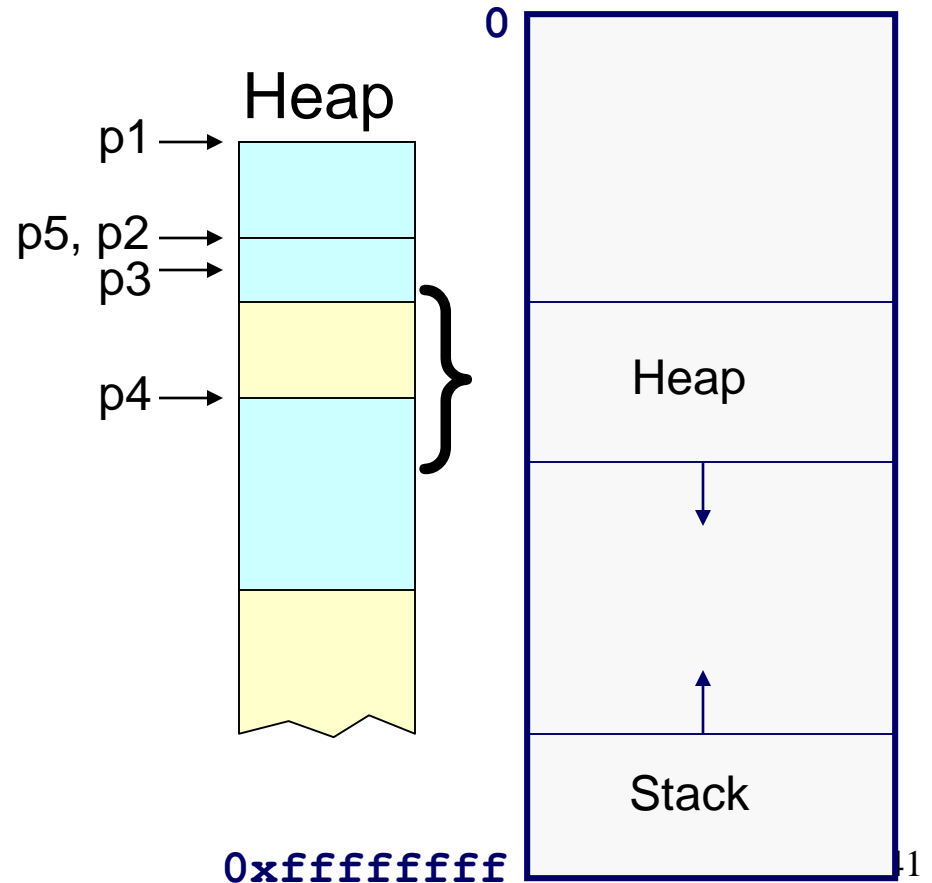


# DMMgr Desired Behavior Demo



DMMgr reused previously freed chunk

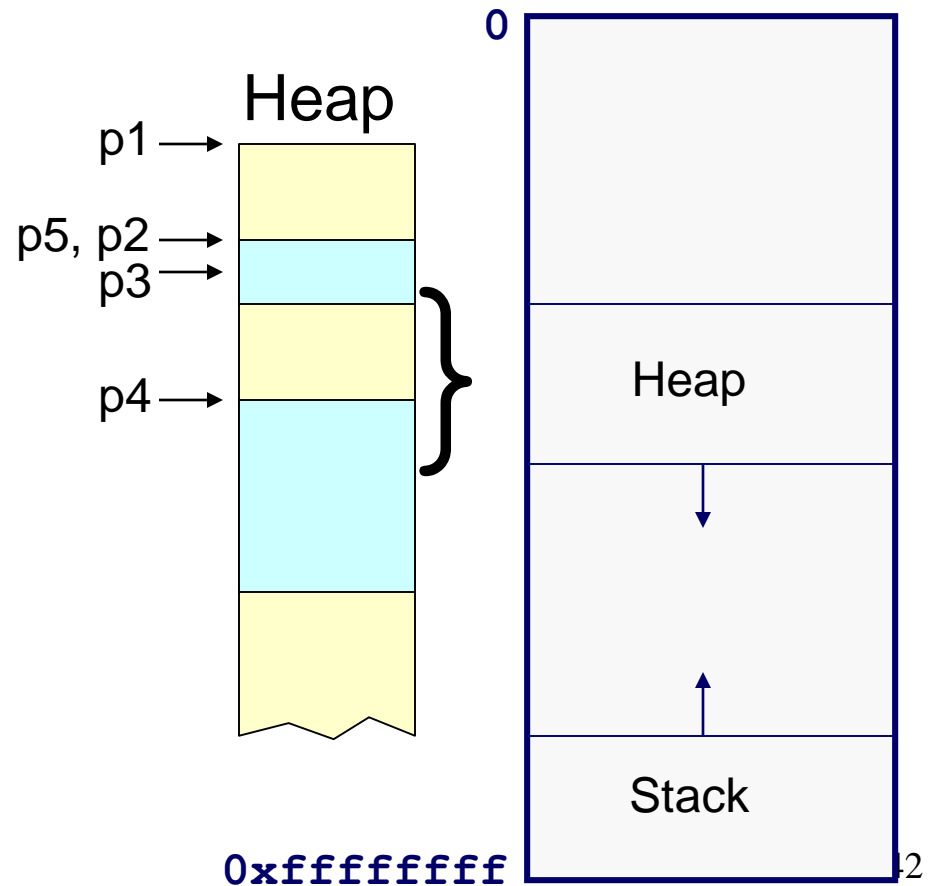
```
char *p1 = malloc(3);  
char *p2 = malloc(1);  
char *p3 = malloc(4);  
free(p2);  
char *p4 = malloc(6);  
free(p3);  
→ char *p5 = malloc(2);  
free(p1);  
free(p4);  
free(p5);
```



# DMMgr Desired Behavior Demo



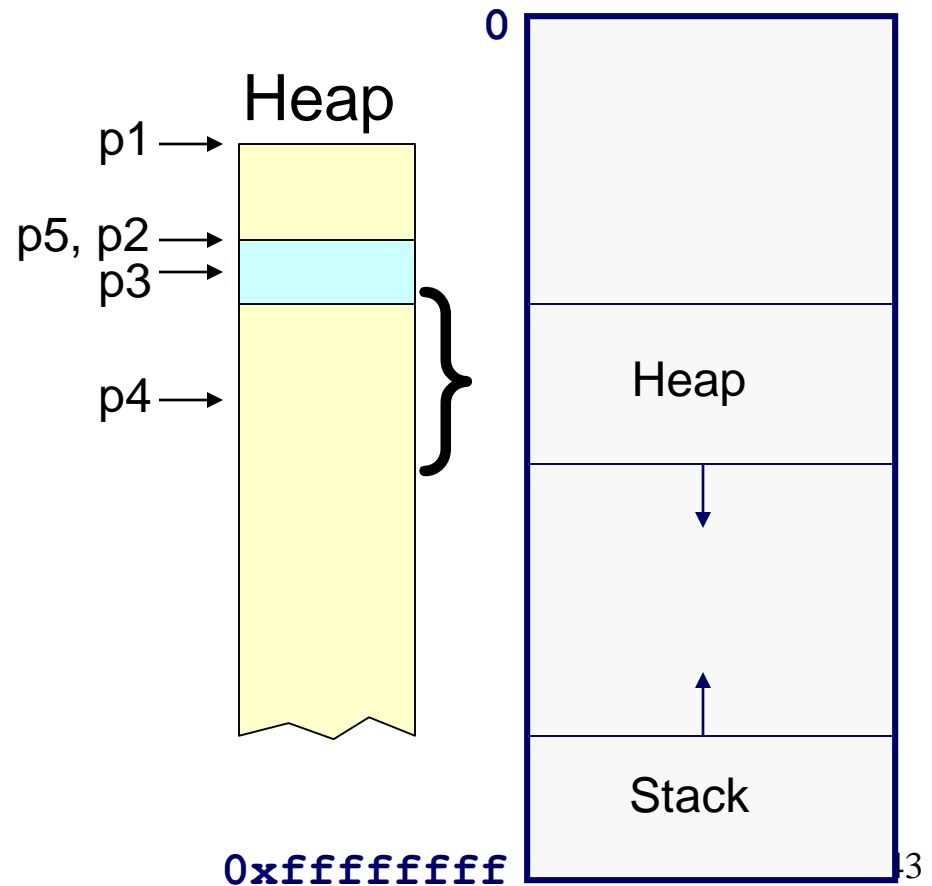
```
char *p1 = malloc(3);  
char *p2 = malloc(1);  
char *p3 = malloc(4);  
free(p2);  
char *p4 = malloc(6);  
free(p3);  
char *p5 = malloc(2);  
→ free(p1);  
free(p4);  
free(p5);
```



# DMMgr Desired Behavior Demo



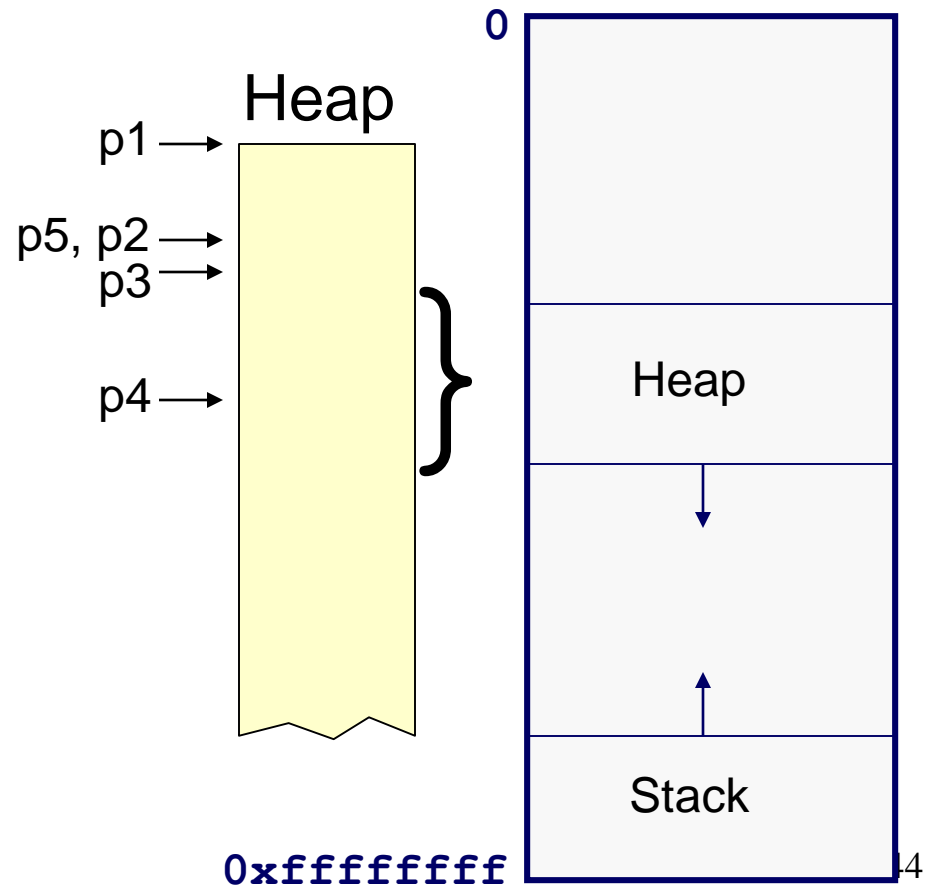
```
char *p1 = malloc(3);  
char *p2 = malloc(1);  
char *p3 = malloc(4);  
free(p2);  
char *p4 = malloc(6);  
free(p3);  
char *p5 = malloc(2);  
free(p1);  
➔ free(p4);  
free(p5);
```



# DMMgr Desired Behavior Demo



```
char *p1 = malloc(3);  
char *p2 = malloc(1);  
char *p3 = malloc(4);  
free(p2);  
char *p4 = malloc(6);  
free(p3);  
char *p5 = malloc(2);  
free(p1);  
free(p4);  
→ free(p5);
```



# DMMgr Desired Behavior Demo



## DMMgr cannot:

- Reorder requests
  - Client may allocate & free in arbitrary order
  - Any allocation may request arbitrary number of bytes
- Move memory chunks to improve performance
  - Client stores addresses
  - Moving a memory chunk would invalidate client pointer!

Some external fragmentation is unavoidable

# Agenda



The need for DMM

DMM using the heap section

DMMgr 1: Minimal implementation

DMMgr 2: Pad implementation

Fragmentation

**DMMgr 3: List implementation**

DMMgr 4: Doubly-linked list implementation

DMMgr 5: Bins implementation

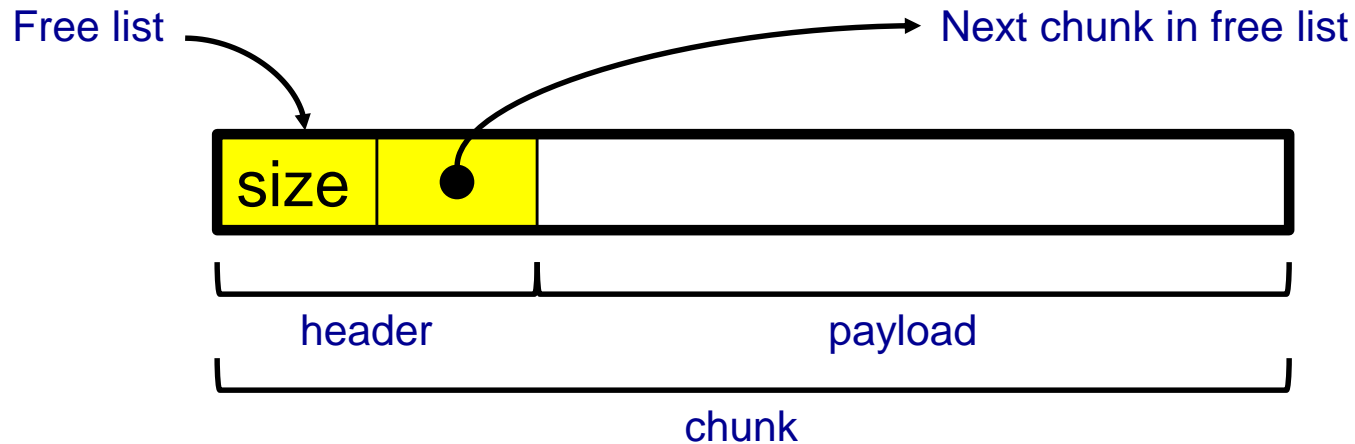
DMM using virtual memory

DMMgr 6: VM implementation



# List Impl

## Data structures



**Free list** contains all free chunks

In order by mem addr

Each chunk contains header & payload

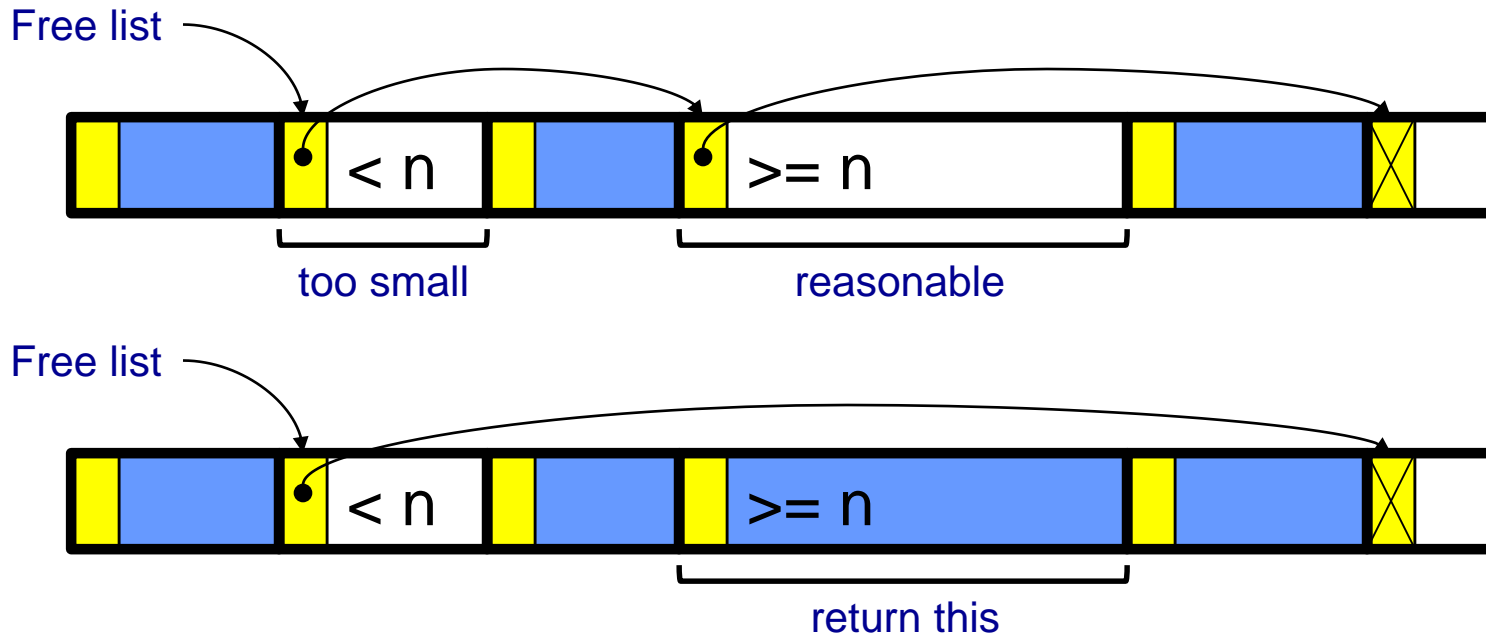
**Payload** is used by client

**Header** contains chunk size & (if free) addr of next chunk in free list

Algorithms (by examples)...



# List Impl: malloc(n) Example 1

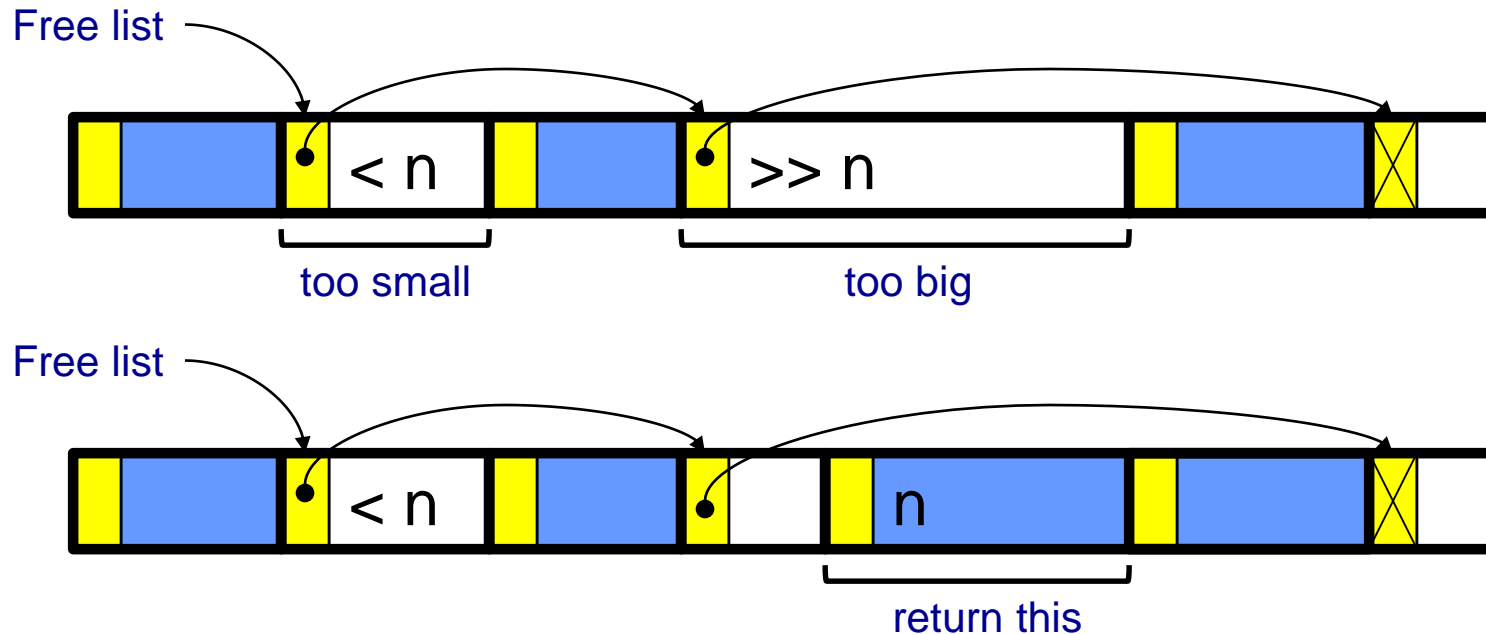


Search list for big-enough chunk  
Note: **first-fit** (not **best-fit**) strategy  
Found & reasonable size =>  
Remove from list and return payload





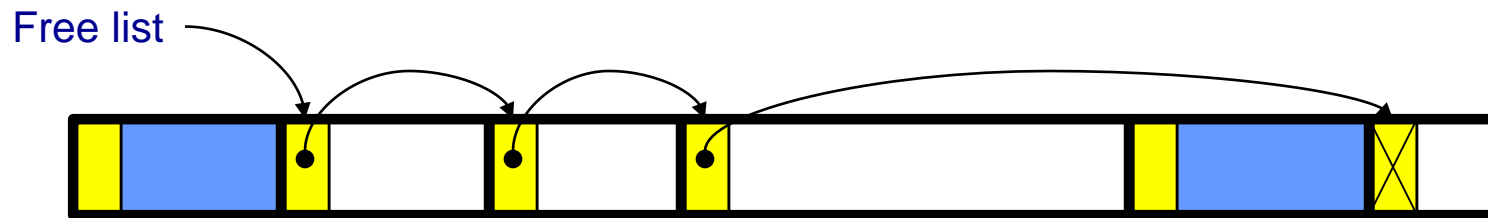
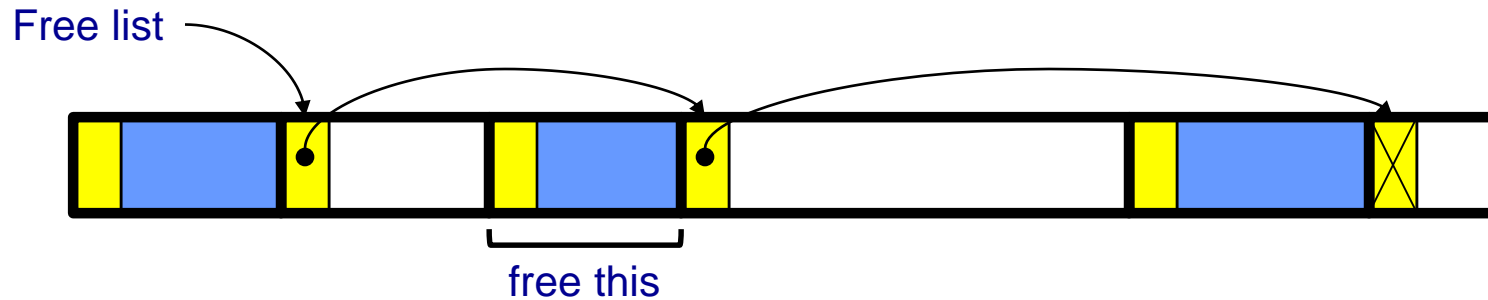
# List Impl: malloc(n) Example 2



Search list for big-enough chunk  
Found & too big =>  
Split chunk, return payload of tail end  
Note: Need not change links

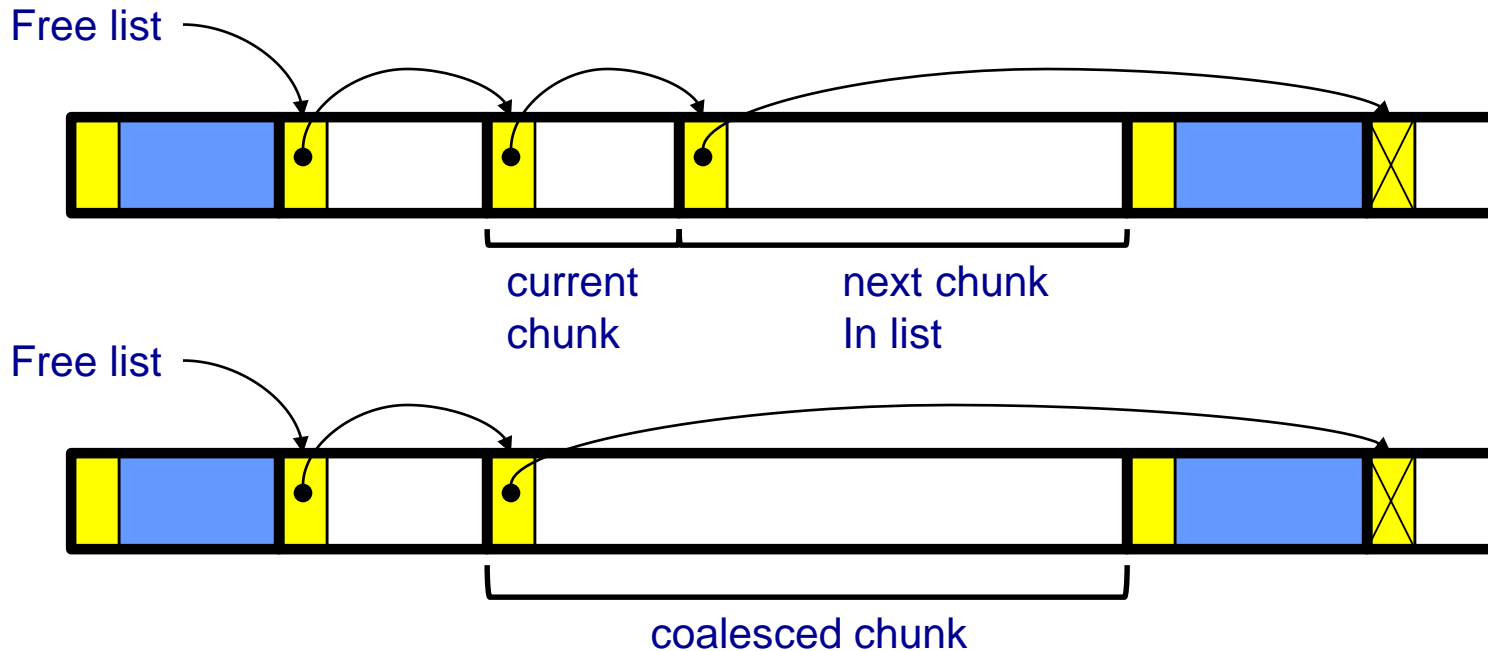


# List Impl: free(p) Example



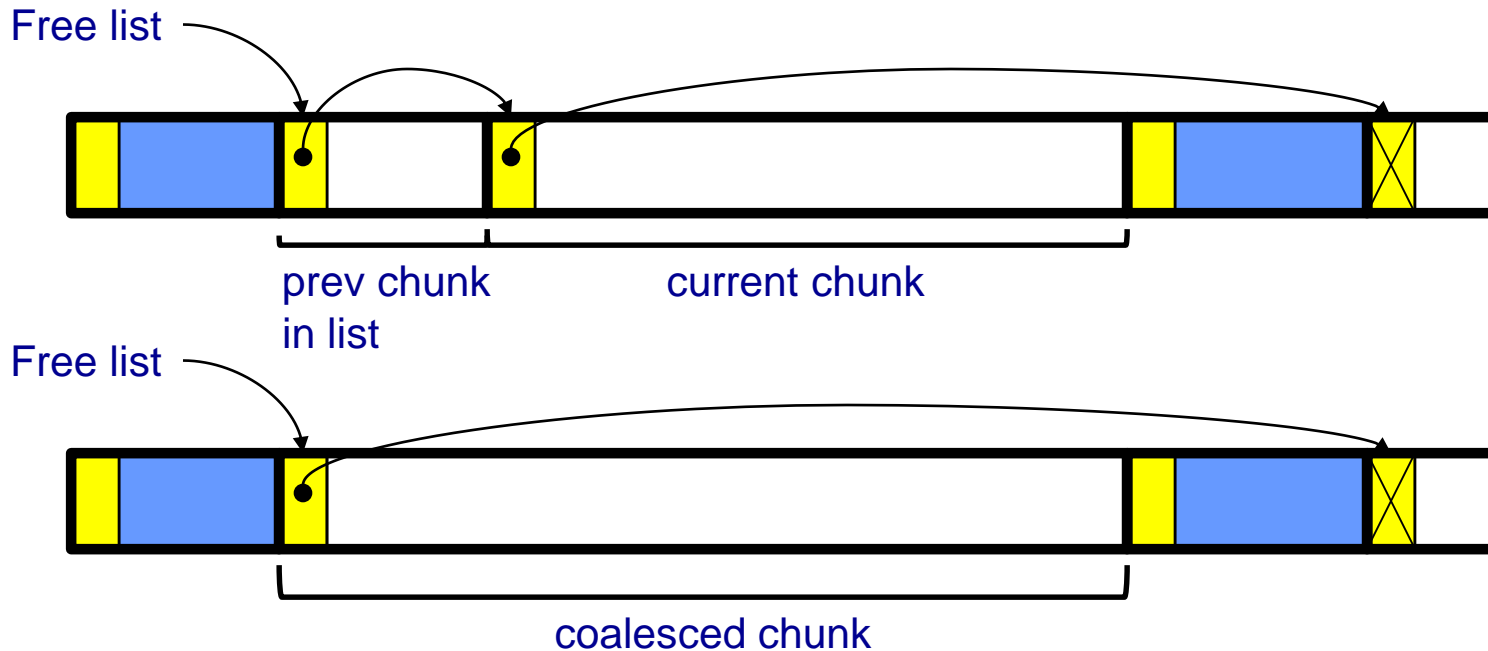
Search list for proper insertion spot  
Insert chunk into list  
(Not finished yet!)

# List Impl: free(p) Example (cont.)



Look at current chunk  
Next chunk in memory == next chunk in list =>  
Remove both chunks from list  
Coalesce  
Insert chunk into list  
(Not finished yet!)

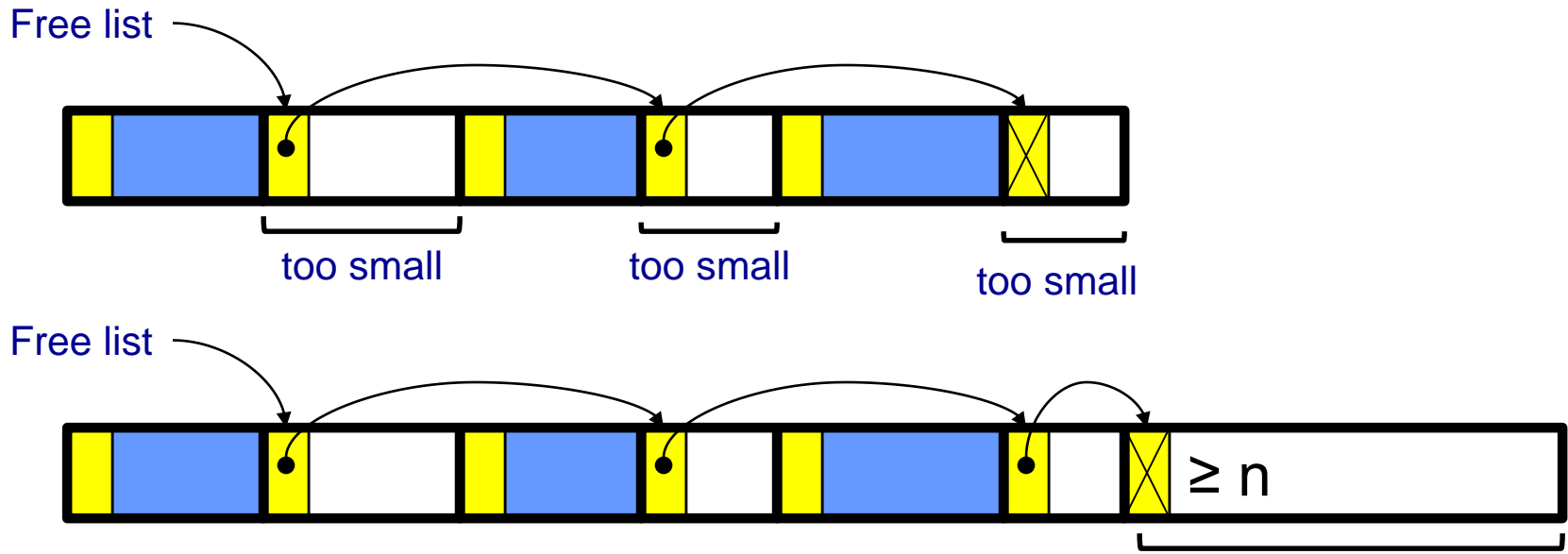
# List Impl: free(p) Example (cont.)



Look at prev chunk in list  
Next in memory == next in list =>  
Remove both chunks from list  
Coalesce  
Insert chunk into list  
(Finished!)



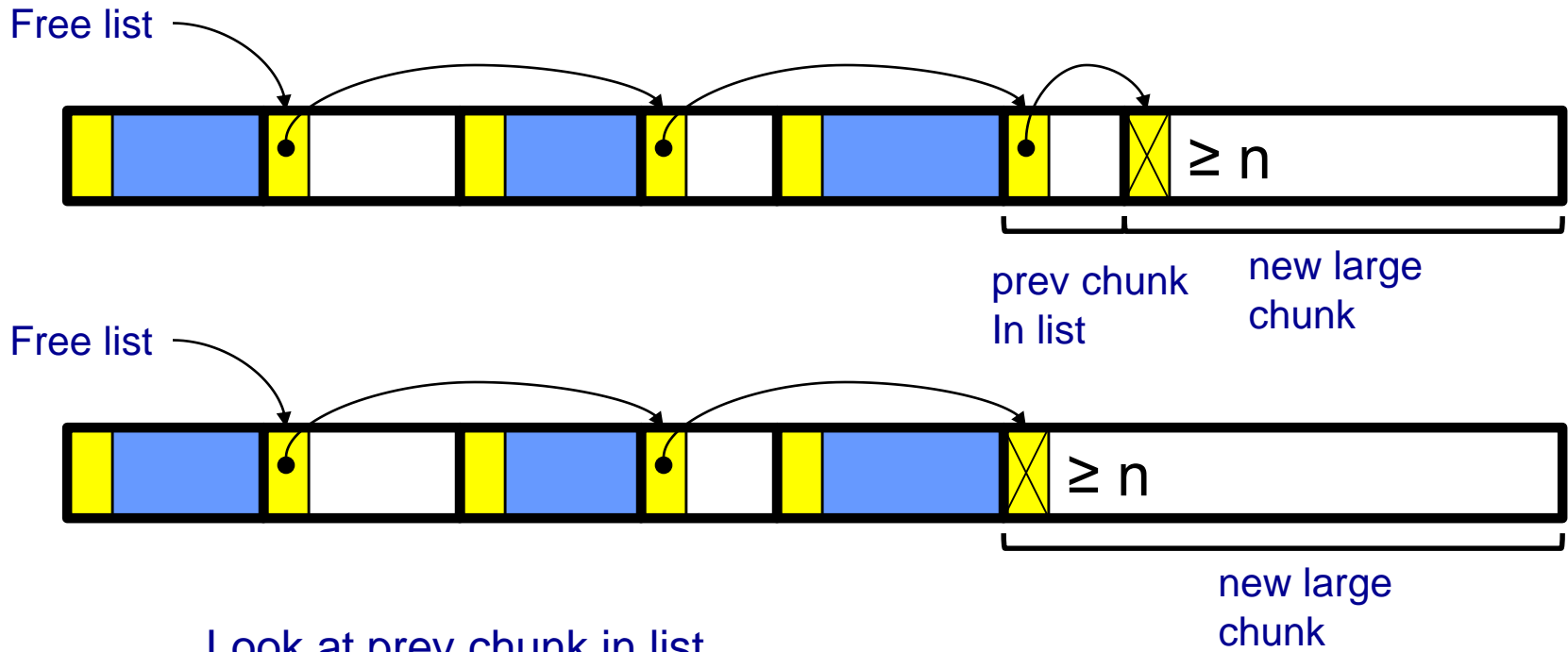
# List Impl: malloc(n) Example 3



Search list for big-enough chunk  
None found =>  
Call `brk()` to increase heap size  
Insert new chunk at end of list  
(Not finished yet!)

new large chunk

# List Impl: malloc(n) Example 3 (cont.)



Look at prev chunk in list  
Next chunk memory == next chunk in list =>  
Remove both chunks from list  
Coalesce  
Insert chunk into list  
Then proceed to use the new chunk, as before  
(Finished!)

# List Impl



Algorithms (see precepts for more precision)

## **malloc (n)**

- Search free list for big-enough chunk
- Chunk found & reasonable size => remove, use
- Chunk found & too big => split, use tail end
- Chunk not found => increase heap size, create new chunk
- New chunk reasonable size => remove, use
- New chunk too big => split, use tail end

## **free (p)**

- Search free list for proper insertion spot
- Insert chunk into free list
- Next chunk in memory also free => remove both, coalesce, insert
- Prev chunk in memory free => remove both, coalesce, insert



# List Impl Performance

## Space

- Some internal & external fragmentation is unavoidable
- Headers are overhead
- Overall: good

## Time: `malloc()`

- Must search free list for big-enough chunk
- Bad:  $O(n)$
- But often acceptable

## Time: `free()`

- Must search free list for insertion spot
- Bad:  $O(n)$
- Often **very** bad



# What's Wrong?



## Problem

- `free()` must traverse (long) free list, so can be (very) slow

## Solution

- Use a doubly-linked list...

# Agenda



The need for DMM

DMM using the heap section

DMMgr 1: Minimal implementation

DMMgr 2: Pad implementation

Fragmentation

DMMgr 3: List implementation

**DMMgr 4: Doubly-linked list implementation**

DMMgr 5: Bins implementation

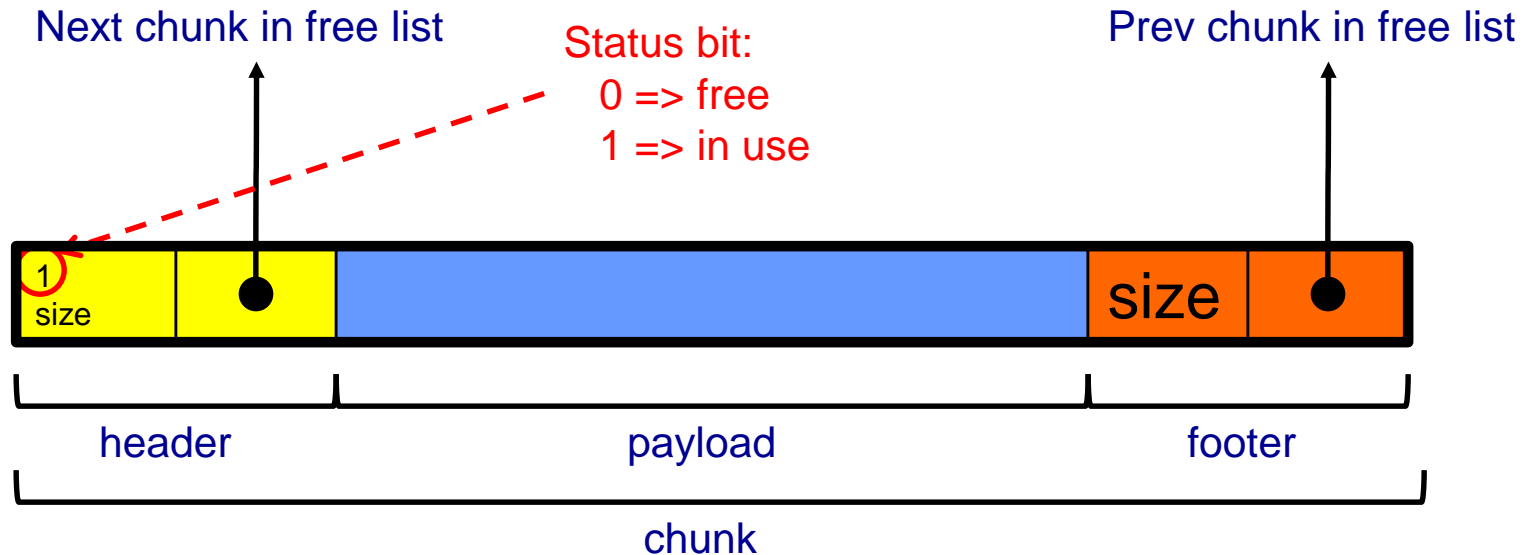
DMM using virtual memory

DMMgr 6: VM implementation



# Doubly-Linked List Impl

## Data structures



**Free list** is doubly-linked

Each chunk contains header, payload, footer

**Payload** is used by client

**Header** contains status bit, chunk size, & (if free) addr of next chunk in list

**Footer** contains redundant chunk size & (if free) addr of prev chunk in list

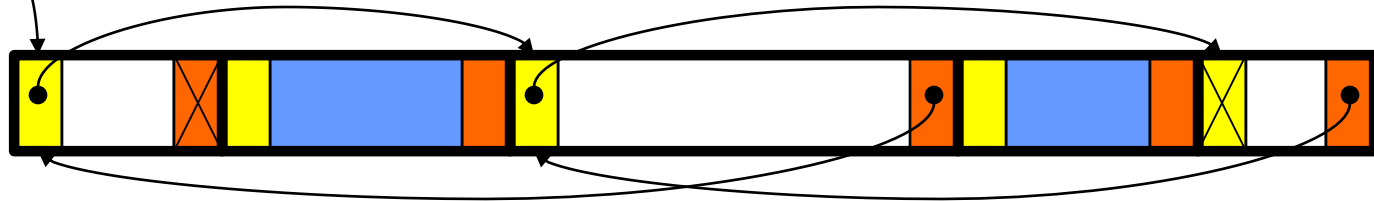
**Free list is unordered**

# Doubly-Linked List Impl



Typical heap during program execution:

Free list





# Doubly-Linked List Impl

Algorithms (see precepts for more precision)

## `malloc(n)`

- Search free list for big-enough chunk
- Chunk found & reasonable size => remove, set status, use
- Chunk found & too big => remove, split, insert tail, set status, use front
- Chunk not found => increase heap size, create new chunk, insert
- New chunk reasonable size => remove, set status, use
- New chunk too big => remove, split, insert tail, set status, use front

# Doubly-Linked List Impl



Algorithms (see precepts for more precision)

**free (p)**

- Set status
- ~~Search free list for proper insertion spot~~
- Insert chunk into free list
- Next chunk in memory also free => remove both, coalesce, insert
- Prev chunk in memory free => remove both, coalesce, insert

# Doubly-Linked List Impl Performance



Consider sub-algorithms of `free ()` ...

Insert chunk into free list

- **Linked list version:** slow
  - Traverse list to find proper spot
- **Doubly-linked list version:** fast
  - Insert at front!

Remove chunk from free list

- **Linked list version:** slow
  - Traverse list to find prev chunk in list
- **Doubly-linked list version:** fast
  - Use backward pointer of current chunk to find prev chunk in list

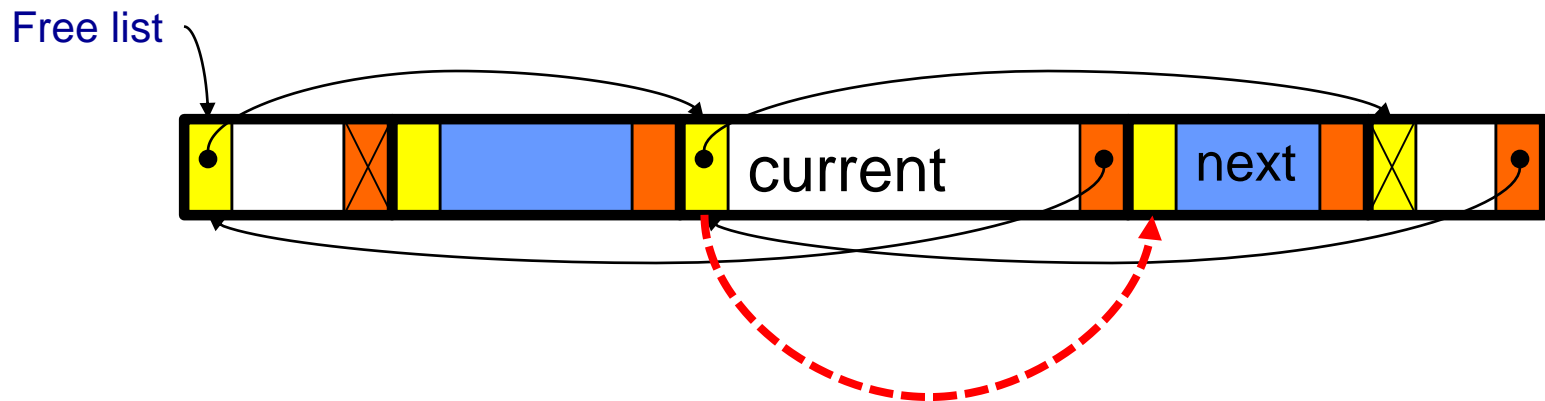


# Doubly-Linked List Impl Performance

Consider sub-algorithms of `free ()` ...

Determine if next chunk in memory is free

- **Linked list version:** slow
  - Traverse free list to see if next chunk in memory is in list
- **Doubly-linked list version:** fast



Use current chunk's size to find next chunk  
Examine status bit in next chunk's header



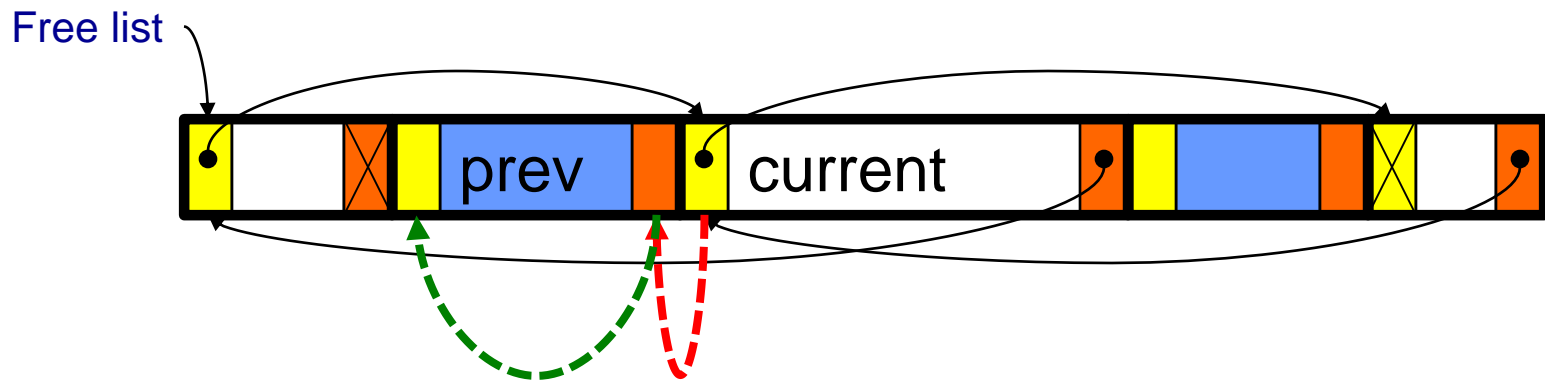


# Doubly-Linked List Impl Performance

Consider sub-algorithms of `free()` ...

Determine if prev chunk in memory is free

- **Linked list version:** slow
  - Traverse free list to see if prev chunk in memory is in list
- **Doubly-linked list version:** fast



Fetch prev chunk's size from its footer  
Do ptr arith to find prev chunk's header  
Examine status bit in prev chunk's header

# Doubly-Linked List Impl Performance



## Observation:

- All sub-algorithms of `free ()` are fast
- `free ()` is fast!



# Doubly-Linked List Impl Performance

## Space

- Some internal & external fragmentation is unavoidable
- Headers & footers are overhead
- Overall: Good

## Time: `free()`

- All steps are fast
- Good:  $O(1)$

## Time: `malloc()`

- Must search free list for big-enough chunk
- Bad:  $O(n)$
- Often acceptable
- Subject to bad worst-case behavior
  - E.g. long free list with big chunks at end

# What's Wrong?



## Problem

- `malloc()` must traverse doubly-linked list, so can be slow

## Solution

- Use multiple doubly-linked lists (bins)...

# Agenda



The need for DMM

DMM using the heap section

DMMgr 1: Minimal implementation

DMMgr 2: Pad implementation

Fragmentation

DMMgr 3: List implementation

DMMgr 4: Doubly-linked list implementation

**DMMgr 5: Bins implementation**

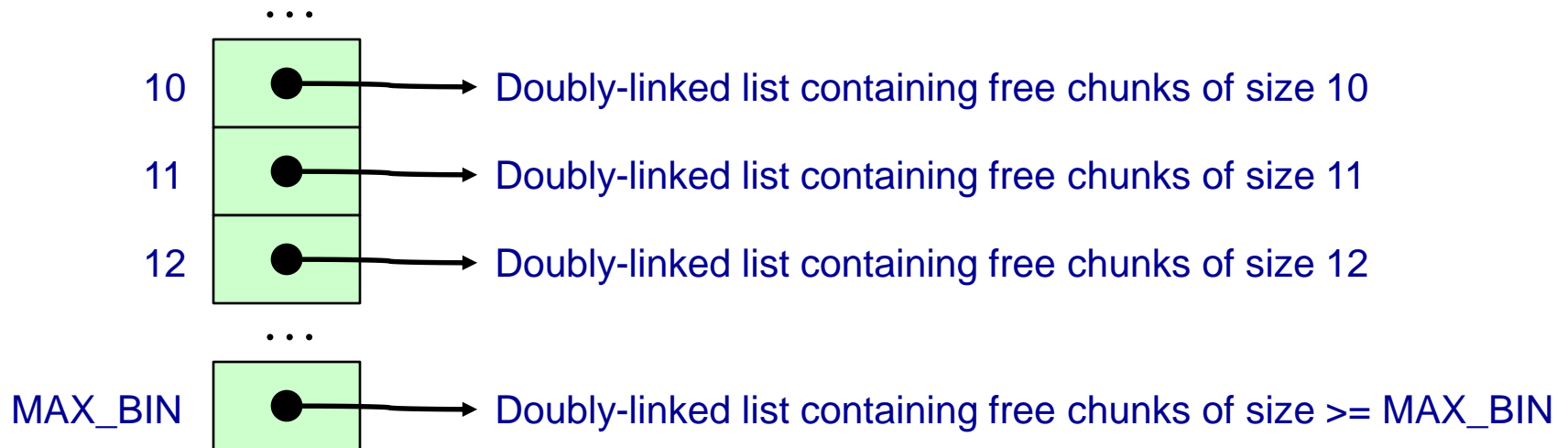
DMM using virtual memory

DMMgr 6: VM implementation



# Bins Impl

## Data structures



Use an array; each element is a **bin**

Each bin is a doubly-linked list of free chunks

As in previous implementation

bin[i] contains free chunks of size i

Exception: Final bin contains chunks of size MAX\_BIN **or larger**

(More elaborate binning schemes are common)



# Bins Impl

Algorithms (see precepts for more precision)

## **malloc(n)**

- Search ~~free list~~ **proper bin(s)** for big-enough chunk
- Chunk found & reasonable size => remove, set status, use
- Chunk found & too big => remove, split, insert tail, set status, use front
- Chunk not found => increase heap size, create new chunk
- New chunk reasonable size => remove, set status, use
- New chunk too big => remove, split, insert tail, set status, use front

## **free(p)**

- Set status
- Insert chunk into ~~free list~~ **proper bin**
- Next chunk in memory also free => remove both, coalesce, insert
- Prev chunk in memory free => remove both, coalesce, insert



# Bins Impl Performance

## Space

- **Pro:** For small chunks, uses **best-fit** (not **first-fit**) strategy
  - Could decrease internal fragmentation and splitting
- **Con:** Some internal & external fragmentation is unavoidable
- **Con:** Headers, footers, bin array are overhead
- **Overall:** good

## Time: `malloc()`

- **Pro:** Binning limits list searching
  - Search for chunk of size  $i$  begins at bin  $i$  and proceeds downward
- **Con:** Could be bad for large chunks (i.e. those in final bin)
  - Performance degrades to that of list version
- **Overall:** good  $O(1)$

## Time: `free()`

- Good:  $O(1)$



# DMMgr Impl Summary (so far)



Implementation	Space	Time
(1) Minimal	Bad	Malloc: Bad Free: Good
(2) Pad	Bad	Malloc: Good Free: Good
(3) List	Good	Malloc: Bad (but could be OK) Free: Bad
(4) Doubly-Linked List	Good	Malloc: Bad (but could be OK) Free: Good
(5) Bins	Good	Malloc: Good Free: Good

Assignment 6: Given (3), compose (4) and (5)

# What's Wrong?



## Observations

- Heap mgr might want to free memory chunks by **unmapping** them rather than **marking** them
  - Minimizes virtual page count
- Heap mgr can call `brk (pBrk-n)` to decrease heap size
  - And thereby unmap heap memory
- But often memory to be unmapped is not at high end of heap!

## Problem

- How can heap mgr unmap memory effectively?

## Solution

- Don't use the heap!

# What's Wrong?



Reprising a previous slide...

Question:

- How to implement `malloc()` and `free()`?
- How to implement a DMMgr?

Answer 1:

- Use the heap section of memory

Answer 2:

- Make use of virtual memory concept...

# Agenda



The need for DMM

DMM using the heap section

DMMgr 1: Minimal implementation

DMMgr 2: Pad implementation

Fragmentation

DMMgr 3: List implementation

DMMgr 4: Doubly-linked list implementation

DMMgr 5: Bins implementation

**DMM using virtual memory**

DMMgr 6: VM implementation



# Unix VM Mapping Functions

Unix allows application programs to map/unmap VM explicitly

```
void *mmap(void *p, size_t n, int prot, int flags, int fd, off_t offset);
```

- Creates a new mapping in the virtual address space of the calling process
- **p**: the starting address for the new mapping
- **n**: the length of the mapping
- If **p** is NULL, then the kernel chooses the address at which to create the mapping; this is the most portable method of creating a new mapping
- On success, returns address of the mapped area

```
int munmap(void *p, size_t n);
```

- Deletes the mappings for the specified address range



# Unix VM Mapping Functions

Typical call of `mmap ()` for allocating memory

```
p = mmap (NULL, n, PROT_READ | PROT_WRITE,  
          MAP_PRIVATE | MAP_ANON, 0, 0);
```

- Asks OS to map a new read/write area of virtual memory containing `n` bytes
- Returns the virtual address of the new area on success, `(void*) -1` on failure

Typical call of `munmap ()`

```
status = munmap (p, n);
```

- Unmaps the area of virtual memory at virtual address `p` consisting of `n` bytes
- Returns 1 on success, 0 on failure

See Bryant & O' Hallaron book and man pages for details

# Agenda



The need for DMM

DMM using the heap section

DMMgr 1: Minimal implementation

DMMgr 2: Pad implementation

Fragmentation

DMMgr 3: List implementation

DMMgr 4: Doubly-linked list implementation

DMMgr 5: Bins implementation

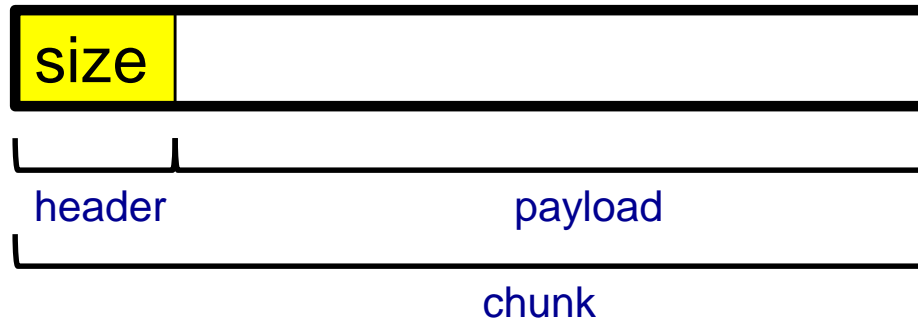
DMM using virtual memory

**DMMgr 6: VM implementation**

# VM Mapping Impl



## Data structures



Each chunk consists of a header and payload  
Each header contains size



# VM Mapping Impl



## Algorithms

```
void *malloc(size_t n)
{ size_t *p;
  if (n == 0) return NULL;
  p = mmap(NULL, n + sizeof(size_t), PROT_READ|PROT_WRITE,
           MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);
  if (p == (void*)-1) return NULL;
  *p = n + sizeof(size_t); /* Store size in header */
  p++; /* Move forward from header to payload */
  return p;
}
```

```
void free(void *p)
{ if (p == NULL) return;
  p--; /* Move backward from payload to header */
  munmap(p, *p);
}
```

# VM Mapping Impl Performance



## Space

- Fragmentation problem is delegated to OS
- Overall: Depends on OS

## Time

- For small chunks
  - One system call (`mmap()`) per call of `malloc()`
  - One system call (`munmap()`) per call of `free()`
  - Overall: poor
- For large chunks
  - `free()` unmaps (large) chunks of memory, and so shrinks page table
  - Overall: maybe good!



# The GNU Implementation

## Observation

- `malloc()` and `free()` on nobel are from the **GNU** (the GNU Software Foundation)

## Question

- How are GNU `malloc()` and `free()` implemented?

## Answer

- For small chunks
  - Use heap (`sbrk()` and `brk()`)
  - Use bins implementation
- For large chunks
  - Use VM directly (`mmap()` and `munmap()`)

# Summary



## The need for DMM

- Unknown object size

## DMM using the heap section

- On Unix: `sbrk ()` and `brk ()`
- Complicated data structures and algorithms
- Good for managing small memory chunks

## DMM using virtual memory

- On Unix: `mmap ()` and `munmap ()`
- Good for managing large memory chunks

See Appendix for additional approaches/refinements

# Appendix: Additional Approaches



Some additional approaches to dynamic memory mgmt...

# Selective Splitting



## Observation

- In previous implementations, `malloc()` splits whenever chosen chunk is too big

## Alternative: **selective splitting**

- Split only when remainder is above some threshold

## Pro

- Reduces external fragmentation

## Con

- Increases internal fragmentation

# Deferred Coalescing



## Observation

- Previous implementations do coalescing whenever possible

## Alternative: **deferred coalescing**

- Wait, and coalesce many chunks at a later time

## Pro

- Handles `malloc(n) ; free() ; malloc(n)` sequences well

## Con

- Complicates algorithms

# Segregated Data



## Observation

- Splitting and coalescing consume lots of overhead

## Problem

- How to eliminate that overhead?

## Solution: **segregated data**

- **Make use of the virtual memory concept...**
- Use bins
- Store each bin's chunks in a distinct (segregated) virtual memory page
- Elaboration...



# Segregated Data



## Segregated data

- Each bin contains chunks of fixed sizes
  - E.g. 32, 64, 128, ...
- All chunks within a bin are from same **virtual memory** page
- **malloc()** never splits! Examples:
  - **malloc(32)** => provide 32
  - **malloc(5)** => provide 32
  - **malloc(100)** => provide 128
- **free()** never coalesces!
  - Free block => examine address, infer virtual memory page, infer bin, insert into that bin

# Segregated Data



## Pros

- Eliminates splitting and coalescing overhead
- Eliminates most meta-data; only forward links required
  - No backward links, sizes, status bits, footers

## Con

- Some usage patterns cause excessive external fragmentation
  - E.g. Only one `malloc(32)` wastes all but 32 bytes of one virtual page



# Segregated Meta-Data

## Observations

- Meta-data (chunk sizes, status flags, links, etc.) are scattered across the heap, interspersed with user data
- Heap mgr often must traverse meta-data

## Problem 1

- User error easily can corrupt meta-data

## Problem 2

- Frequent traversal of meta-data can cause excessive page faults (poor locality)

## Solution: **segregated meta-data**

- **Make use of the virtual memory concept...**
- Store meta-data in a distinct (segregated) virtual memory page from user data