Princeton University
Computer Science 217: Introduction to Programming Systems

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
12	brk()	Move the program break, thus changing the amount of memory allocated to the HEAP
12	sbrk()	(Variant of previous)
9	mmap()	Map a virtual memory page
11	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
- Number of objects needed not known in advance
  (e.g., how many elements of linked list or tree?)
- Unknown object size
  (e.g., how large should the array be, in hash table?)

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

7

Option 1: Automatic Freeing

Run-time system frees unneeded memory
- Java, Python, …
- **Garbage collection**

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

Original Car object can't be accessed

Pros:
- Easy for programmer
- Fewer bugs
- Simpler interfaces between modules
- Fewer bugs

Cons:
- Performed constantly ⇒ overhead
- Performed periodically ⇒ unexpected pauses

(these days, high-performance garbage collectors minimize overhead and pause latency)

8

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

9

Conclusion:

Program in a safe, garbage-collected language!

(not in C)

Use unsafe languages with manual memory management (such as C)

only for low-level programs where the overhead or latency of garbage collection is intolerable

such as: OS kernels, device drivers

All right then, let's see how manual memory management works in C

10

C memory allocation library

Standard C dynamic-memory-management functions:

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

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

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

11

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)

12

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

13

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

14

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

Note: minimal interface (good!)

15

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

16

Minimal Impl

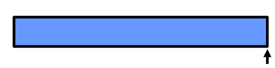
Data structures
- One word: remember the current value of program break

Algorithms (by examples)…

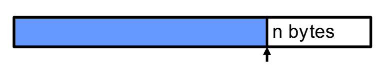
17

Minimal Impl malloc(n) Example

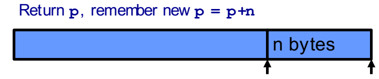
Remember the current program break (p) (initialize using `sbrk(0)`)



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



Return `p`, remember new `p = p+n`



18

Minimal Impl free(p) Example

Do nothing!

Minimal Impl

Algorithms

```
static void *current_break;

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

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


Minimal Impl Performance

Performance (general case)

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

What's Wrong?

Problem

- `malloc()` executes a system call every time

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

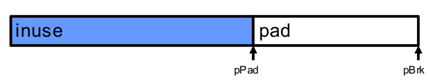
DMMgr 5: Bins implementation

DMM using virtual memory

DMMgr 6: VM implementation

Pad Impl

Data structures

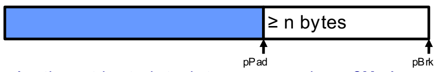


- `pBrk`: address of end of heap (i.e. the program break)
- `pPad`: address of beginning of pad

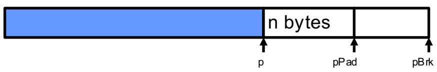
```
char *pPad = NULL;
char *pBrk = NULL;
```

Algorithms (by examples)…

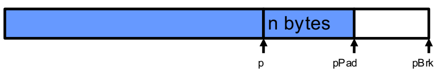
Pad Impl malloc(n) Example 1



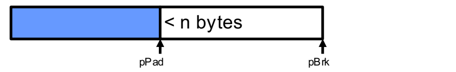
Are there at least **n** bytes between **pPad** and **pBrk**? **Yes**!
Save **pPad** as **p**; add **n** to **pPad**



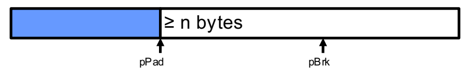
Return **p**



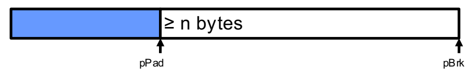
Pad Impl malloc(n) Example 2



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



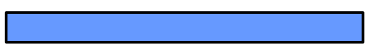
Set **pBrk** to new program break



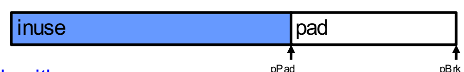
Proceed as previously!

Pad Impl free(p) Example

Do nothing!



Pad Impl



Algorithms

```
void *malloc(size_t n)
{  enum {MIN_ALLOC = 8192};
   char *p;
   char *pNewBrk;
   if (pBrk == NULL)
   {  pBrk = sbrk(0);
      pPad = pBrk;
   }
   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;
}
```

```
void free(void *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

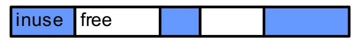
Solution

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

Agenda

Fragmentation

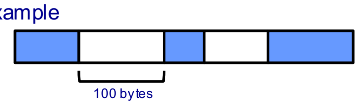
At any given time, some heap memory chunks are in use, some are marked "free"



DMMgr must be concerned about **fragmentation**…

Internal Fragmentation

Internal fragmentation: waste **within** chunks
Example



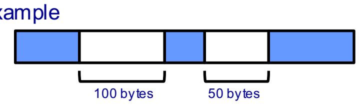
Client asks for 90 bytes
DMMgr provides chunk of size 100 bytes
10 bytes wasted

Generally
- Program asks for n bytes
- DMMgr provides chunk of size n+Δ bytes
- Δ 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

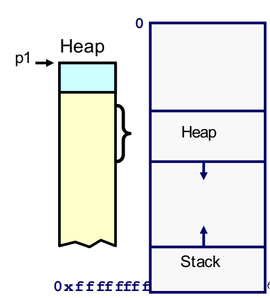
Space efficiency ⇒
- DMMgr should reduce external fragmentation

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);
```

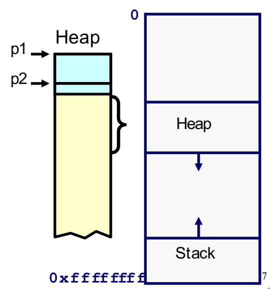

DMMgr Desired Behavior Demo

```
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);
```



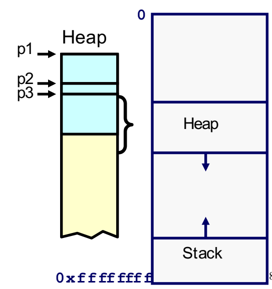
DMMgr Desired Behavior Demo

```
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

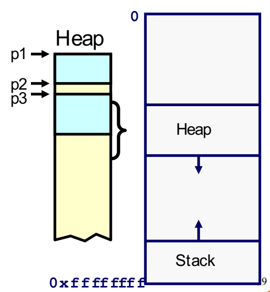
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



DMMgr Desired Behavior Demo

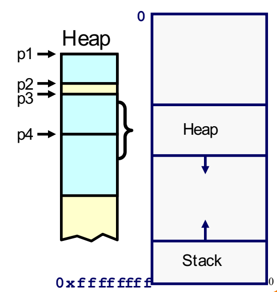
External fragmentation occurred

```
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

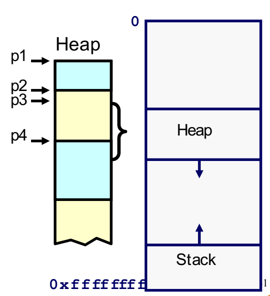
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



DMMgr Desired Behavior Demo

DMMgr coalesced two free chunks

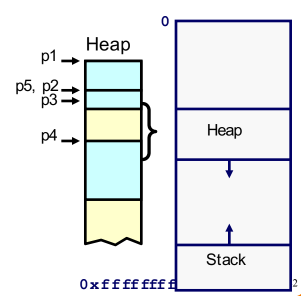
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



DMMgr Desired Behavior Demo

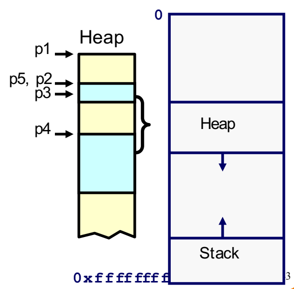
DMMgr reused previously freed chunk

```
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);
```



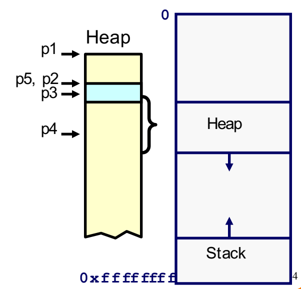
DMMgr Desired Behavior Demo

```
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);
```



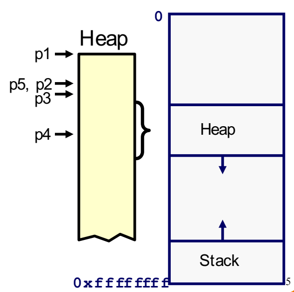
DMMgr Desired Behavior Demo

```
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

46

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

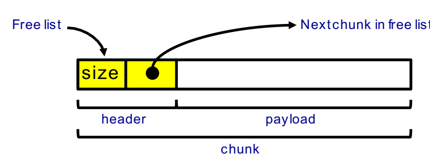
DMM using virtual memory

DMMgr 6: VM implementation

47

List Impl

Data structures

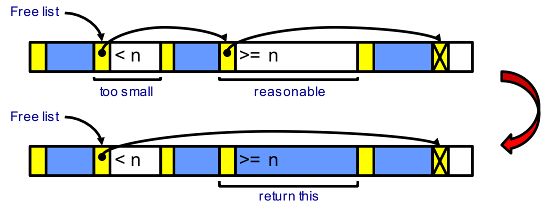


Free list contains all free chunks
In order by mem addr
Each chunk contains header & payload
Payload is used by client
Header contains chunk size & (if free) addr of next chunk in free list

Algorithms (by examples)...

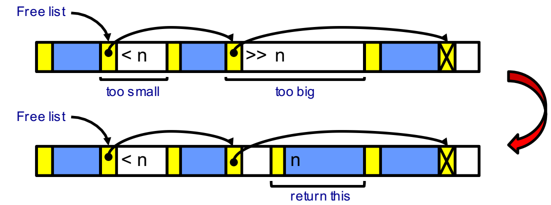
48

List Impl: malloc(n) Example 1



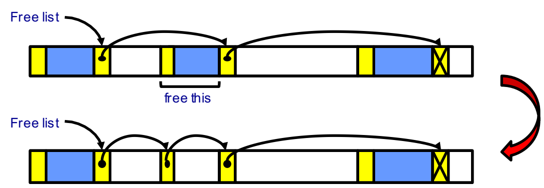
Search list for big-enough chunk
Note: **first-fit** (not **best-fit**) strategy
Found & reasonable size ⇒
Remove from list and return payload

List Impl: malloc(n) Example 2



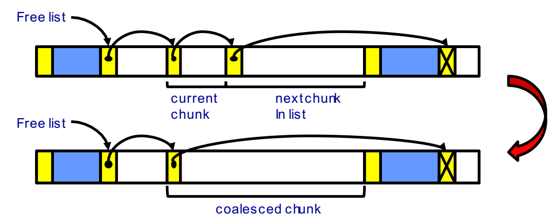
Search list for big-enough chunk
Found & too big ⇒
Split chunk, return payload of tail end
Note: Need not change links

List Impl: free(p) Example



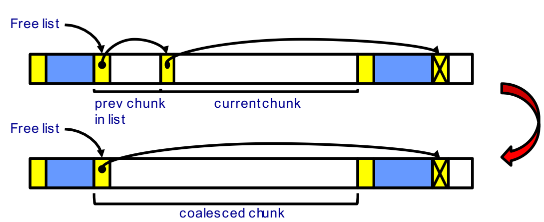
Search list for proper insertion spot
Insert chunk into list
(Not finished yet!)

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



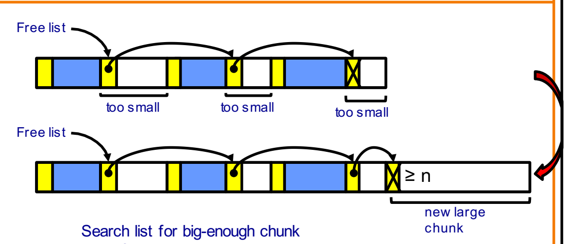
Look at current chunk
Next chunk in memory == next chunk in list ⇒
Remove both chunks from list
Coalesce
Insert chunk into list
(Not finished yet!)

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



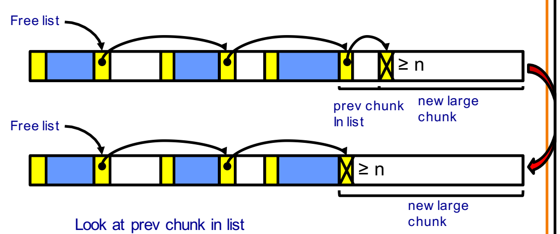
Look at prev chunk in list
Next in memory == next in list ⇒
Remove both chunks from list
Coalesce
Insert chunk into list
(Finished!)

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!)

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!)

55

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

56

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

57

What's Wrong?

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

Solution
- Use a doubly-linked list…

58

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

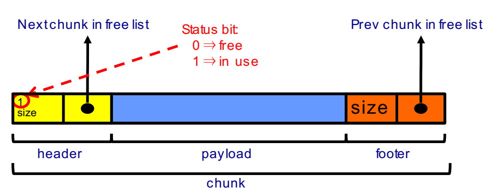
DMM using virtual memory

DMMgr 6: VM implementation

59

Doubly-Linked List Impl

Data structures

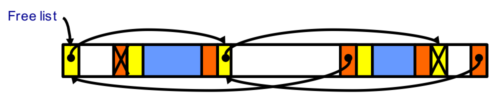


Free list is doubly-linked
Each chunk contains header, payload, footer
Payload is used by client
Header contains status bit, chunk size, & (if free) addr of next chunk in list
Footer contains redundant chunk size & (if free) addr of prev chunk in list
Free list is unordered (i.e., chunks in free list not ordered by address)

60

Doubly-Linked List Impl

Typical heap during program execution:



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

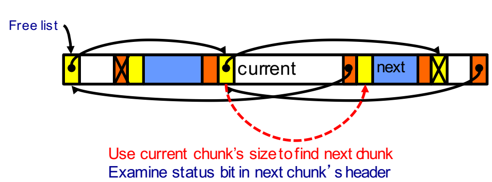
- **Linked list version**: slow
  - Traverse list to find prev chunk in list
- **Doubly-linked list version**: fast
  - Use backward pointer of current chunk to find prev chunk in list

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

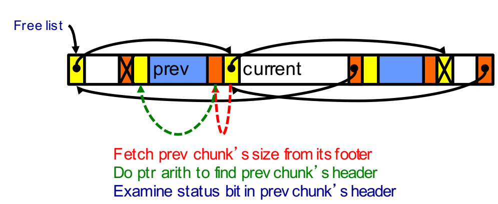


Doubly-Linked List Impl Performance

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

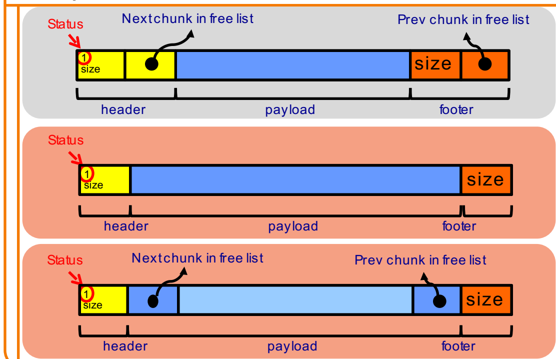
Determine if prev chunk *in memory* is free

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



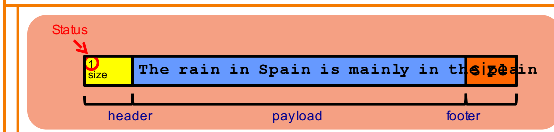
Using payload space for management

or, only free chunks need to be in the free-list



This trick is NOT part of assignment 6!

Another use for the extra size field: error checking




```
char *s = (char *)malloc(32);
   . . .
strcpy(s, "The rain in Spain is mainly in the plain.");
   . . .
printf("%s\n", s);
free(s);
```


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

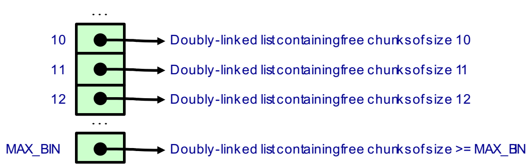
DMMgr 5: Bins implementation

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)

73

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

74

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)

75

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)

76

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

77

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

78

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 0 on success, -1 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

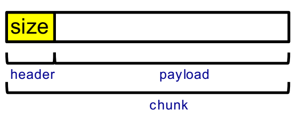
DMMgr 5: Bins implementation

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: **bad**
- For large chunks
  - `free()` unmaps (large) chunks of memory, and so shrinks page table
  - Overall: **good**

The GNU Implementation

Observation

- `malloc()` and `free()` on CourseLab 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 dynamic memory management
- 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

85

Appendix: Additional Approaches

Some additional approaches to dynamic memory mgmt…

86

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

87

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

88

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…

89

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

90

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 Metadata

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

Problem 1
- User error easily can corrupt metadata

Problem 2
- Frequent traversal of meta-data can cause excessive page faults (poor locality)

Solution: **segregated metadata**
- **Make use of the virtual memory concept…**
- Store metadata in a distinct (segregated) virtual memory page from user data