Dynamic Memory Allocation

Jinyang Li

based on Tiger Wang’s slides
**Why dynamic memory allocation?**

Allocation size is unknown until the program runs (at runtime).

```c
#define MAXN 15213
int array[MAXN];

int main(void)
{
    int i, n;
    scanf("%d", &n);
    if (n > MAXN)
        app_error("Input file too big");
    for (i = 0; i < n; i++)
        scanf("%d", &array[i]);
    exit(0);
}
```
Why dynamic memory allocation?

Allocation size is unknown until the program runs (at runtime).

```c
int main(void)
{
    int *array, i, n;
    scanf("%d", &n);
    array = (int *)malloc(n * sizeof(int));
    for (i = 0; i < n; i++)
        scanf("%d", &array[i]);
    exit(0);
}
```
Dynamic allocation on heap

Question: can one dynamically allocate memory on stack?
Dynamic allocation on heap

Question: Is it possible to dynamically allocate memory on stack?

Answer: Yes, but space is freed upon function return

```c
#include <stdlib.h>
void *alloca(size_t size);

void func(int n) {
    array = alloca(n);
}
```

Not good practice!
Dynamic allocation on heap

Question: How to allocate memory on heap?
Dynamic allocation on heap

Question: How to allocate memory on heap?

Ask OS for allocation on the heap via system calls

`void *sbrk(intptr_t size);`

It increases the top of heap by “size” and returns a pointer to the base of new storage. The “size” can be a negative number.
Dynamic allocation on heap

Question: How to allocate memory on heap?

Ask OS for allocation on the heap via system calls

```c
void *sbrk(intptr_t size);
```

It increases the top of heap by “size” and returns a pointer to the base of new storage. The “size” can be a negative number.

```c
p = sbrk(1024) //allocate 1KB
```

Diagram:
- User stack
- Shared libraries
- Runtime heap
  - Read/write segment (.data, .bss)
  - Read-only segment (.init, .text, .rodata)
  - Unused
- %rsp (stack pointer)
- brk

Loaded from the executable file
Question: How to allocate memory on heap?

Ask OS for allocation on the heap via system calls

```c
void *sbrk(intptr_t size);
```

It increases the top of heap by “size” and returns a pointer to the base of new storage. The “size” can be a negative number.

```c
p = sbrk(1024) //allocate 1KB
sbrk(-1024) //free p
```
Dynamic allocation on heap

Question: How to allocate memory on heap?

Ask OS for allocation on the heap via system calls

```c
void *sbrk(intptr_t size);
```

Issue I – can only free the memory on the top of heap

```c
p1 = sbrk(1024)  // allocate 1KB
p2 = sbrk(4096)  // allocate 4KB
```

How to free p1?

![Diagram showing allocation on heap and memory segments](image)
Dynamic allocation on heap

Question: How to allocate memory on heap?

Ask OS for allocation on the heap via system calls

```c
void *sbrk(intptr_t size);
```

Issue I – can only free the memory on the top of heap

Issue II – system call has high performance cost > 10X
Dynamic allocation on heap

Question: How to efficiently allocate memory on heap?

Basic idea – request a large of memory region from heap once, then manage this memory region by itself. → allocator in the library
Memory Allocator

Assumption in this lecture

– At the beginning, the allocator requests enough memory with sbrk

Goal

– Efficiently utilize acquired memory with high throughput
  • high throughput – how many mallocs / frees can be done per second
  • high utilization – fraction of allocated size / total heap size
Memory Allocator

Assumed behavior of applications:
- Issue an arbitrary sequence of malloc/free
- Argument of free must be the return value of a previous malloc
- No double free

Restrictions on the allocator:
- Once allocated, cannot be moved
Questions

1. (Basic book-keeping) How to keep track which bytes are free and which are not?

2. (Allocation decision) Which free chunk to allocate?

3. (API restriction) free is only given a pointer, how to find out the allocated chunk size?
How to bookkeep? Strawman #1

• Structure heap as n 1KB chunks + n metadata

```c
#define CHUNKSIZE
typedef chunk char[CHUNKSIZE];
char *bitmap;
chunk *chunks;
size_t n_chunks;

void init() {
  n_chunks = 1<<10;
  sbrk(n_chunks*CHUNKSIZE + n_chunks/8);
  bitmap = heap_hi()+1 - n_chunks/8;
  chunks = (chunk *)heap_lo();
}
```
How to bookkeep? Strawman #1

```c
void *malloc(size_t sz) {
    assert(sz < CHUNKSIZE);
    size_t i = 0;
    for (; i < n_chunks; i++) {
        if !bitmap_get_pos(bitmap, i)
            break; //found a free chunk
    }

    if (i == n_chunks) //did not find a free chunk
        return NULL;

    bitmap_set_pos(bitmap, i);
    return (void *)&chunk[i];
}
```

chunks

p = malloc(1000);

bitmap
How to bookkeep? Strawman #1

- Problem with strawman?
  - cannot malloc more than a chunk at a time
  - cannot malloc less than a chunk

```c
void free(void *p) {
  i = ((char *)p - (char *)chunks)/CHUNKSIZE;
  bitmap_unset_pos(bitmap, i);
}
```
How to bookkeep? Other Strawmans

• How to support a variable number of variable-sized chunks?
  – Idea #1: use a hash table to map address → [chunk size, status]
  – Idea #2: use a linked list in which each node stores [address, chunk size, status] information.

Problems of strawmans?
Implementing a hash table and linked list requires use of a dynamic memory allocator!
How to implement a "linked list" without use of malloc
Implicit list

- Embed chunk metadata in the chunks
  - Chunk has a header storing size and status
  - Payload is 16-byte aligned
    → Chunk size (metadata+payload) is multiple of 16
    → Header must be also aligned to 16 bytes

<table>
<thead>
<tr>
<th>chunk size</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-byte padding</td>
<td>(16 bytes)</td>
</tr>
<tr>
<td>Payload</td>
<td></td>
</tr>
<tr>
<td>Padding (optional)</td>
<td></td>
</tr>
</tbody>
</table>

status: allocated or free

allocated: size_and_status & 0x1L
size: size_and_status & ~0x1L
Implicit list

Embed chunk metadata in the chunks
- Chunk has a header storing size and status
- Payload is 16-byte aligned

\[ p = \text{malloc}(1024) \]
Implicit list

Embed chunk metadata in the chunks
- Chunk has a header storing size and status
- Payload is 16-byte aligned

\[
p = \text{malloc}(1)
\]
Implicit list

Embed chunk metadata in the chunks
- Chunk has a header storing size and status
- Payload is 16-byte aligned

\[ p = \text{malloc}(1) \]
How to traverse an implicit list

typedef struct {
    unsigned long size_and_status;
    unsigned long padding;
} header;

void traverse_implicit_list() {
    header *curr = (header *)heap_lo();
    while ((char *)curr < heap_high()) {
        bool allocated = get_status(curr);
        size_t csz = get_chunksz(curr);
        curr = (header *)((char *)curr + csz);
    }
}

bool get_status(header *h) {
    return h->size_and_status & 0x1L;
}

size_t get_chunksize(header *h) {
    return h->size_and_status & ~(0x1L);
}
Placing allocated blocks

\[
p1 = \text{malloc}(8) \\
p2 = \text{malloc}(24) \\
p3 = \text{malloc}(56) \\
p4 = \text{malloc}(8) \\
p5 = \text{malloc}(24) \\
p6 = \text{malloc}(56)
\]
Placing allocated blocks

\[ p1 = \text{malloc}(8) \]
\[ p2 = \text{malloc}(24) \]
\[ p3 = \text{malloc}(56) \]
\[ p4 = \text{malloc}(8) \]
\[ p5 = \text{malloc}(24) \]
\[ p6 = \text{malloc}(56) \]
\[ \text{free}(p2) \]
\[ \text{free}(p4) \]
\[ \text{free}(p6) \]
Placing allocated blocks

p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
p7 = malloc(8)

First fit – Search list from beginning, choose first free block that fits
Placing allocated blocks

First fit – Search list from beginning, choose first free block that fits

Issue: cause “splinters/fragments” at beginning of the buffer

\[
\begin{align*}
p1 &= \text{malloc}(8) \\
p2 &= \text{malloc}(24) \\
p3 &= \text{malloc}(56) \\
p4 &= \text{malloc}(8) \\
p5 &= \text{malloc}(24) \\
p6 &= \text{malloc}(56) \\
\text{free}(p2) \\
\text{free}(p4) \\
\text{free}(p6) \\
p7 &= \text{malloc}(8) \\
\end{align*}
\]
Placing allocated blocks

p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
p7 = malloc(8)

Best fit – choose the free block with the closest size that fits
Placing allocated blocks

p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
p7 = malloc(8)

Best fit – choose the free block with the closest size that fits
Placing allocated blocks

$p1 = malloc(8)$
$p2 = malloc(24)$
$p3 = malloc(56)$
$p4 = malloc(8)$
$p5 = malloc(24)$
$p6 = malloc(56)$
free(p2)
free(p4)
free(p6)
$p7 = malloc(8)$

Best fit – Search list from beginning, choose first free block that fits

Best fit keeps fragments small, but typically run slower than first fit.
Placing allocated blocks

```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
p7 = malloc(8)
p8 = malloc(56)
```

Next fit – like first-fit, but search list from the location where previous search left off.
Placing allocated blocks

Next fit – like first-fit, but search list from the location where previous search left off.

Next fit runs faster than first fit, but fragmentation is worse.

\[
p_1 = \text{malloc}(8) \\
p_2 = \text{malloc}(24) \\
p_3 = \text{malloc}(56) \\
p_4 = \text{malloc}(8) \\
p_5 = \text{malloc}(24) \\
p_6 = \text{malloc}(56) \\
\text{free}(p_2) \\
\text{free}(p_4) \\
\text{free}(p_6) \\
p_7 = \text{malloc}(8) \\
p_8 = \text{malloc}(56)
\]
Placing allocated blocks

\[ p_1 = \text{malloc}(8) \]
\[ p_2 = \text{malloc}(24) \]
\[ p_3 = \text{malloc}(56) \]
\[ p_4 = \text{malloc}(8) \]
\[ p_5 = \text{malloc}(24) \]
\[ p_6 = \text{malloc}(56) \]
\[ \text{free}(p_2) \]
\[ \text{free}(p_4) \]
\[ \text{free}(p_6) \]
\[ p_7 = \text{malloc}(24) \]
\[ p_8 = \text{malloc}(24) \]
Splitting free block

p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(16)
p6 = malloc(48)
free(p2)
free(p4)
free(p6)
p7 = malloc(16)
p8 = malloc(16)
Splitting free block

\[
p_1 = \text{malloc}(8)  
p_2 = \text{malloc}(24)  
p_3 = \text{malloc}(56)  
p_4 = \text{malloc}(8)  
p_5 = \text{malloc}(16)  
p_6 = \text{malloc}(48)  
\text{free}(p_2)  
\text{free}(p_4)  
\text{free}(p_6)  
p_7 = \text{malloc}(16)  
p_8 = \text{malloc}(16)
\]
Splitting free block

```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(16)
p6 = malloc(48)
free(p2)
free(p4)
free(p6)
p7 = malloc(16)
p8 = malloc(16)
```
Splitting free block

p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(16)
p6 = malloc(48)
free(p2)
free(p4)
free(p6)
p7 = malloc(16)
p8 = malloc(16)
Coalescing free blocks

p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(16)
p6 = malloc(48)
free(p2)
free(p4)
free(p6)
free(p5)
Coalescing free blocks

p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(16)
p6 = malloc(48)
free(p2)
free(p4)
free(p6)
free(p5)
Coalescing free blocks

p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(16)
p6 = malloc(48)
free(p2)
free(p4)
free(p6)
free(p5)

Coalescing with next block
Coalescing free blocks

\[ p_1 = \text{malloc}(8) \]
\[ p_2 = \text{malloc}(24) \]
\[ p_3 = \text{malloc}(56) \]
\[ p_4 = \text{malloc}(8) \]
\[ p_5 = \text{malloc}(16) \]
\[ p_6 = \text{malloc}(48) \]
\[ \text{free}(p_2) \]
\[ \text{free}(p_4) \]
\[ \text{free}(p_6) \]
\[ \text{free}(p_5) \]

Coalescing with next block

How to coalesce with previous block?
Coalescing free blocks

p1 = malloc(8)

p2 = malloc(24)

p3 = malloc(56)

p4 = malloc(8)

p5 = malloc(16)

p6 = malloc(48)

free(p2)

free(p4)

free(p6)

free(p5)

Coalescing with next block
How to coalesce with previous block?
-- search from start?
Coalescing free blocks

Embed chunk metadata in the chunks
- Chunk has a header storing size and status
- Payload is 16-byte aligned
  - Easiest way to align is to make chunk size, header and footer all 16-byte aligned.

```
p = malloc(1024)
0x421 8B padding
1KB payload
0x421 8B padding
```

```
header (16 bytes)
Payload
Padding (optional)
footer (16 bytes)
```
Coalescing free blocks

\[
p1 = \text{malloc}(8) \\
p2 = \text{malloc}(24) \\
p3 = \text{malloc}(56) \\
p4 = \text{malloc}(8) \\
p5 = \text{malloc}(16) \\
p6 = \text{malloc}(48) \\
\text{free}(p2) \\
\text{free}(p4) \\
\text{free}(p6) \\
\text{free}(p5)
\]
Coalescing free blocks

p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(16)
p6 = malloc(48)
free(p2)
free(p4)
free(p6)
free(p5)
Explicit free lists

Problems of implicit list:
- Allocation time is linear in # of total (free and allocated) chunks

Explicit free list:
- Maintain a linked list of free chunks only.
Explicit free list

- Question: do we need next/prev fields for allocated blocks?
  Answer: No. We do not need to traverse allocated blocks only. We can still traverse all blocks (free and allocated) as in the case of implicit list.

- Question: what’s the minimal size of a chunk?
  Answer: 16 (header) + 16 (footer) + 8 (next pointer) + 8 (previous pointer) = 48 bytes
How to traverse an explicit list

typedef struct free_header {  
    header common_header;  
    struct free_header *next;  
    struct free_header *prev;  
} free_header;

free_header *freelist;  
void init() {  
    //starts with a list of one free chunk  
}  
void traverse_explicit_list() {  
    free_hdr *f = freelist;  
    while (f!=NULL) {  
        bool allocated = get_status(f->common_header.size_n_status);  
        size_t csz = get_chunksz(f->common_header.size_n_status);  
        f = f->next;  
    }  
}
void *malloc(size_t size) {
    free_header *f;
    f = get_freechunk(freelist, size);

    delete_from_linked_list(freelist, f);
    set_status(f->header.size_n_status); //set footer too if there’s one

    return (void *)(((char *)f)+sizeof(f->header));
}
Segregated list

• Idea: keep multiple freelists
  – each freelist contains chunks of similar sizes
Segregated list

Free lists

{16}  
{32-48}  
{64 – 128}

Determine size class

First fit

fit?

Yes

Remove and split free block

Insert the fragment

No

Search in next free list
Buddy System

Adopted by Linux kernel and jemalloc

This lecture
– A simplified binary buddy allocator
Binary buddy system
Binary buddy system

Split
- Split exactly in half
Binary buddy system

Split
- Split exactly in half
- Each half is the buddy of the other

Address
- Block of size $2^n$ begin at memory addresses where the $n$ least significant bits are zero
- When a block of size $2^{n+1}$ is split into two blocks of size $2^n$, the addresses of these two blocks will differ in exactly one bit, bit $n$.

If a block of size $2^n$ begins at address $addr$, what is its buddy address and size?
Binary buddy system

Split
- Split exactly in half
- Each half is the buddy of the other

Address
- Block of size $2^n$ begin at memory addresses where the $n$ least significant bits are zero
- When a block of size $2^{n+1}$ is split into two blocks of size $2^n$, the addresses of these two blocks will differ in exactly one bit, bit $n$.

If a block of size $2^n$ begins at address $addr$, what is its buddy address and size?
$addr$ of buddy = $addr$ $\oplus$ $(1 << n)$
Binary buddy system

Split
- Split exactly in half
- Each half is the buddy of the other

Address
- Block of size $2^n$ begin at memory addresses where the $n$ least significant bits are zero
- When a block of size $2^{n+1}$ is split into two blocks of size $2^n$, the addresses of these two blocks will differ in exactly one bit, bit $n$.

Combine
- only combine a block with its buddy
Buddy system

Observation

(100)\textsubscript{2}

(110)\textsubscript{2}

Each list has the same size of blocks which is a power of 2.

Determine size class

First fit

fit?

Yes

No

Search in next free list

recursively split free block in half until fit

Insert the fragment
Buddy system

\[ p = \text{malloc}(1) \]

Step 1. search in \( 2^6 \) list

Step 2. recursive split

Each list has the same size of blocks which is a power of 2.
Buddy system

\[ p = \text{malloc}(1) \]

Step 1. search in \( 2^6 \) list

Step 2. recursive split

Free lists

\[ 2^4 \quad 2^5 \quad 2^6 \]

\((x000000)_2\)  \((x100000)_2\)

Determine size class

First fit

fit?  

Yes  

recursively split free block in half until fit

Insert the fragment

No  

Search in next free list
Buddy system

\[ p = \text{malloc}(1) \]

Step 1. search in \(2^6\) list

Step 2. recursive split

Free lists

Determine size class

First fit

fit? Yes

recursively split free block in half until fit

Insert the fragment

No

Search in next free list
Buddy system

\[ p = \text{malloc}(1) \]

Step 1. search in \( 2^6 \) list

Step 2. recursive split

\[
\begin{array}{c}
(x000000) \\
(x010000) \\
(x100000) \\
\end{array}
\]
Buddy system

$p = \text{malloc}(1)$

Step 1. search in $2^6$ list
Step 2. recursive split
Step 3. insert free blocks into the list
Step 4. return, $p$ is $(x000000)_2$
Buddy system

\[ p = \text{malloc}(1) \]

\[ \text{free}(p) \]

Free lists

- \( 2^4 \)
- \( 2^5 \)
- \( 2^6 \)

Determine size class

First fit

fit? 

- Yes
  - 
  - recursively split free block in half until fit
  - Insert the fragment

- No
  - Search in next free list
Buddy system

\[ p = \text{malloc}(1) \]
\[ \text{free}(p) \]

p’s address is \( (x000000)_2 \), size is 16B (no footer in this example)

→ p’s buddy is 16 B block begins at \( x000000 \uparrow (1<<4) \) which is \( (x010000)_2 \)
Buddy system

\[ p = \text{malloc}(1) \]

\[ \text{free}(p) \]

p’s address is \((x000000)_2\), size is 32B (no footer in this example)

\( \rightarrow \) p’s buddy is 32 B block begins at \(x000000 \wedge (1<<5)\)
which is \((x100000)_2\)
p = malloc(1)
free(p)

p’s address is (x000000)_2, size is 32B

→ p’s next block address is p + 32B which is (x100000)_2