Dynamic Memory Allocation

Jinyang Li

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

typedef struct node {
    int val;
    struct node *next;
} node;

void list_insert(node **head, int v) {
    node *np = malloc(sizeof(node));
    np->next = head;
    np->val = v;
    *head = np;
}

int main(void) {
    node *head = NULL;
    int n = atoi(argv[1]);
    for (int i = 0; i < n; i++) {
        list_insert(&head, i);
    }
}

How many nodes to allocate is only known at runtime (when the program executes)
Dynamic allocation on heap

Question: can one dynamically allocate memory on stack?

- 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
Dynamic allocation on heap

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

Answer: Yes, but space is freed upon function return

```c
void list_insert(node *head, int v) {
    node n;
    node *np = &n;
    np->next = head;
    np->val = v;
    *head = np;
}
```
Dynamic allocation on heap

Question: How to allocate memory on heap?

- 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.
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
```
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.

```
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

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

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

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

How to free p1?
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: user program asks a large memory region from OS once, then manages this memory region by itself (using a “malloc” library)
How to implement a memory allocator?

API:
- void* malloc(size_t size);
- void free(void *ptr);

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
How to implement a 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, space cannot be moved around
Malloc design challenges

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 1<<10;
typedef char[CHUNKSIZE] chunk;
char *bitmap;
chunk *chunks;
size_t n_chunks;

void init() {
  n_chunks = 128;
  sbrk(n_chunks*sizeof(chunk)+ n_chunks/8);
  chunks = (chunk *)heap_lo();
  bitmap = heap_lo() + n_chunks *CHUNKSIZE;
}
```

Assume allocator asks for enough memory from OS in the beginning
**How to bookkeep? Strawman #1**

```c
void* malloc(size_t sz) {
    // find out # of chunks needed to fit sz bytes
    csz = ...;

    // find csz consecutive free chunks according to bitmap
    int i = find_consecutive_chunks(bitmap);

    // return NULL if did not find csz free consecutive chunks
    if (i < 0)
        return NULL;

    // set bitmap at positions i, i+1, ... i+csz-1
    bitmap_set_pos(bitmap, i, csz);
    return (void *)&chunks[i];
}
```
How to bookkeep? Strawman #1

```c
void free(void *p) {
    i = ((char *)p - (char *)chunks)/sizeof(chunk);
    bitmap_clear_pos(bitmap, i); //how many bits to clear??
}
```

- **Problem with strawman?**
  - `free` does not know how many chunks allocated
  - wasted space within a chunk (internal fragmentation)
  - wasted space for non-consecutive chunks (external fragmentation)
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 “list” without use of malloc
### Implicit list

- Each chunk contains metadata + payload
  - Metadata (chunk header) stores chunk size and status
- Alignment requirement: 16-byte (aka the starting address of payload must be multiples of 16).
  - Make header 16-byte in size
  - Make chunk multiples of 16 in size.

![Diagram of chunk structure]

- chunk size
- status (allocated or free)
- header (16 bytes)
- Payload
Implicit list: chunk format

chunk size = 32+16
status = 1

Payload

header (16 bytes)
n
payload (32 bytes)

\[ p = \text{malloc}(20) \]
Implicit list: heap traversal

How to advance from one chunk to the next?
Which chunk to allocate?

\[
p_1 = \text{malloc}(8) \\
p_2 = \text{malloc}(24) \\
p_3 = \text{malloc}(56) \\
p_4 = \text{malloc}(8) \\
p_5 = \text{malloc}(24) \\
p_6 = \text{malloc}(56) \\
\]
Where to place an allocation?

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

- 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
First fit

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

Downside: cause fragmentation at beginning of the heap
Best fit

- Best fit – choose the free block with the closest size that fits

```c
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

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

Downside: run slower than first fit.
Next fit

- `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 starts from where the previous search left off.
Next fit

- Like first-fit, but search starts from where the previous search left off.

```
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 starts from where the previous search left off.

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

```
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)
```
Splitting a free block

... 

\[ p9 = \text{malloc}(16) \]
Splitting a free block

... p9 = malloc(16)
Coalescing a free block with its next free neighbor

... free(p10)
Coalescing a free block with its next free neighbor

... free(p10)

addr increases

heap

header
allocated payload
padding
free block

size=32
status=0

16B Payload

size=64
status=0

48B Payload
Coalescing a free block with its next free neighbor

... free(p10)

How to coalesce with the previous free block
typedef struct {
    size_t size;
    size_t allocated;
} header_t;

int mm_init();
void *mm_malloc(size_t size);
void mm_free(void *p);
**implicit list implementation**

```c
size_t hdr_size = sizeof(header_t);

int init() {
    return 0; // start with empty heap
}

void *mm_malloc(size_t size) {
    size = align(size);
    size_t csz = size + hdr_size;
    header_t *h = first_fit(csz); // find a free chunk
    if (!h) {
        h = ask_os_for_chunk(csz);
    } else {
        split(h, csz); // split if necessary
    }
    ... // set chunk status to be allocated
}
```
implicit list implementation

```c
header_t *payload2header(void *p) {
}

void *mm_free(void *p) {
    header_t *h = payload2header(p);
    ... //set chunk status to be free
    coalesce(h);
}
```
Use footer to coalesce with previous block

- Duplicate header information into the footer

<table>
<thead>
<tr>
<th>Payload</th>
<th>size</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>header (16 bytes)</td>
</tr>
<tr>
<td></td>
<td>footer (16 bytes)</td>
<td></td>
</tr>
</tbody>
</table>

**Design 1:** super easy

<table>
<thead>
<tr>
<th>Payload</th>
<th>size</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>padding</td>
</tr>
<tr>
<td></td>
<td>size</td>
<td></td>
</tr>
</tbody>
</table>

**Design 2:** less header overhead

- Use LSB for status
Coalescing prev and next blocks

... free(p10)

prev chunk

next chunk

addr increases

p10

heap

previous chunk

16B Payload

size=48
status=0

size=48
status=0

size=48
status=1

16B Payload

size=48
status=0

size=64
status=1

size=64
status=0

32B payload

size=64
status=0
Coalescing prev and next blocks

... free(p10)

addr increases

heap

size=160
status=0

128B Payload

size=160
status=0
Recap: malloc using implicit list

Design without footer

Design with footer
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 chain together allocated blocks. 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: 8 (header) + 8 (footer) + 8 (next pointer) + 8 (previous pointer) = 32 bytes
Explicit list: types, globals, initialization

typedef struct {
    size_t size;
    size_t status;
} hdr_t;

typedef struct free_hdr_t {
    hdr_t header;
    struct free_hdr *next;
    struct free_hdr *prev;
} free_hdr_t;

free_hdr_t *freelist;

void init() {
    freelist = NULL;
}

void *malloc(size_t size) {
    size = align(size);
    size_t csz = sizeof(hdr_t) + size;   //size of chunk to allocate

    free_hdr_t *h = first_fit(csz);
    if (!h) {
        h = ask_os_for_chunk(csz);
    } else {
        split(h);                      //split if necessary
    }
    return (char *)h + sizeof(hdr_t);  //return pointer to payload
}
Explicit list: find a chunk

void list_insert(free_hdr_t **head, free_hdr_t *node) {
  ...
}

void list_delete(free_hdr_t **head, free_hdr_t *node) {
  ...
}

//find a chunk in freelist whose size is >= csz
//detach the found chunk from freelist
free_hdr_t *
first_fit(size_t csz) {
  free_hdr_t *curr = freelist;
    
  while (curr) {
    if (curr->header.size) >= sz) {
      list_delete(&freelist, curr);
      return curr;
    }
    curr = curr->next;
  }
  return NULL;
Explicit list: free

```c
void free(void *p) {
    header *h = payload2header(p);
    init_free_chunk((free_hdr_t *)h, h->size);

    coalesce(h);
    list_insert(&freelist, h);
}

void init_free_chunk(free_hdr_t *h, size_t sz) {
    h->next = h->prev = NULL;
    h->status = 0;
    h->size = size;
}

void coalesce(free_hdr_t *h) {
    header_t *next = next_chunk((header_t)h);
    if (!next->status) {
        list_delete(&freelist, (free_hdr_t *)next);
        h->size += next->size;
    }
}
```
Explicit list with footer

typedef size_t hdr_t;

typedef struct free_hdr_t {
    hdr_t header;
    struct free_hdr *next;
    struct free_hdr *prev;
} free_hdr_t;

bool get_status(hdr_t *h) {
    return (*h) & 0x1L;
}

size_t get_size(hdr_t *h) {
    return (*h) & ~0x1L;
}

void set_size_status(hdr_t *h, size_t sz, bool status) {
    *h = sz | status;
}

void set_status(header *h, bool status){
    ...
}

void set_size(header *h, size_t sz) {
    ...
}
Segregated list

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

#define NLISTS 3
free_hdr* freelists[NLISTS];
size_t size_classes[NLISTS] = {32, 128, -1};

int which_freelist(size_t s) {
    int ind = 0;
    while (s > size_classes[ind])
        ind++;
    return ind;
}

void init() {
    for (int i = 0; i < NLISTS; i++) {
        freelists[i] = NULL;
    }
}
Segregated list: allocation

freelists:

(0, 32] (32, 128] (128, ∞)

Determine size class

First fit

found?

Yes

Remove and split free block
determine size class of split block

Insert split block into the appropriate list

No

Search in next free list
Segregated list: free

freelists:

(0, 32] (32, 128] (128, ∞)

next block is free?

Yes

remove next block from its freelist and merge

prev block is free?

Yes

remove prev block from its freelist and merge

No

determine size class

Insert block into the appropriate list
Buddy System

• A special case of segregated list
  – each freelist has \textit{id}entity\textit{ally}-\textit{sized} blocks
  – block sizes are powers of 2

• Advantage over a normal segregated list?
  – Less search time (no need to search within a freelist)
  – Less coalescing time

• Adopted by Linux kernel and jemalloc
Simple binary buddy system

Initialize:
• assume heap starts at 00...00 (real heap address is just a constant offset from 00...00)
Binary buddy system: allocate

\[ p = \text{malloc}(16000); \]

Recursive split in half until having the right size
- insert free buddy into appropriate freelist

Addresses of buddies at size \(2^m\) differ in exactly 1-bit at position \(m\) (from right)
Binary buddy system: allocate

\[ p = \text{malloc}(16000); \]

Recursive split in half until having the right size
- insert free buddy into appropriate freelist
Binary buddy system: free

```c
free(p);
```

Recursively merge block with buddy

1. Calculate addr of buddy block, determine buddy status

Question: given addr `a` of block with size $2^m$, how to calculate its buddy’s address?

```
(a ^ (1<<m))
```

any bit XOR 0 = unchanged
any bit XOR 1 = flipped
Binary buddy system: free

free(p);

If buddy is free:
2. Detach free buddy from its list
3. Combine with current block
Binary buddy system: free

free(p);

Repeat to merge with larger buddy
Insert final block into appropriate freelist