Memory Management with `mmap`

What if we use `mmap` instead of `malloc` always?

- **Wasteful**
  - *low utilization*
  - need 16 bytes, get 4096

- **Slow**
  - *low throughput*
  - have to interact with kernel every time, and those 4096 bytes are all zeroed

- **Complicated**
  - have to remember the size to unmap
Process Memory Layout

- Kernel virtual memory
- Stack (created at runtime)
- Shared libraries (code and data)
- Heap
- Read+write data (.data, .bss)
- Read-only data (.text, .rodata)
- Unused

Inaccessible to user code

%rsp
brk

0x400000
Memory Management with sbrk

```c
#include <unistd.h>

void *sbrk(intptr_t increment);
```

Grows the **program break**, a.k.a. brk, and returns the old program break

Effectively, allocates `increment` bytes

**Do not use sbrk** in a program that also uses `malloc` or anything that calls `malloc` (such as `printf`)
Memory Management with \texttt{sbrk}

What if we use \texttt{sbrk} instead of \texttt{malloc} always?

\begin{itemize}
  \item[✓] Economical  \quad \textbf{good utilization}, at first need 16 bytes, get 16
  \item[✗] Somewhat slow  \quad \textbf{somehat low throughput}
    have to interact with kernel every time
  \item[✗] Complicated
    have to remember the size to unsbrk(?)
  \item[✗] Inexpressive  \quad \textbf{low utilization} when done with data
    at best, can free last chunk allocated
\end{itemize}
# Standard C Allocation

```c
#include <stdlib.h>

void *malloc(size_t size);
void free(void *p);

void *calloc(size_t count, size_t size);
void *realloc(void *p, size_t new_size);
```

**malloc** allocates at least `size` bytes

**free** accepts a pointer (just once) from **malloc**

  behind the scenes: **mmap** or **sbrk**, maybe **munmap**

**calloc** is multiply, then **malloc**, then **bzero**

**realloc** is **malloc**, then **memcpy**, then **free**

maybe with a shortcut
Allocation Example

\[ p1 = \text{malloc}(4) \]
\[ p2 = \text{malloc}(5) \]
\[ p3 = \text{malloc}(6) \]
\[ \text{free}(p2) \]
\[ p4 = \text{malloc}(2) \]
Allocation: Application Side

Rights:

• Call freely interleave `malloc` and `free`

Responsibilities:

• Must write to only allocated (not-yet-freed) blocks
• Must call `free` only once on each `malloc` result
• Must call `free` enough to limit memory use
Allocation: Allocator Side

Rights:

• Can pick arbitrary virtual addresses
  within alignment constraints

Responsibilities:

• Must accept any size request
• Must accept any number of requests
• Must return non-overlapping blocks
• Must not write to allocated (not-yet-freed) blocks
• Must respond immediately (i.e., can’t reorder requests)
Allocation: Performance Goals

**Utilization** — use as few pages as possible

measure as \(\text{aggregate payload} \over \text{pages used}\)

- \(\text{malloc}(n) \Rightarrow \text{payload size } n\)
- Sum of \(n\) not yet freed = \(\text{aggregate payload}\)

**Throughput** — \(\text{malloc/free}\) as fast as possible

measure as \(\text{number of operations performed} \over \text{seconds used}\)
Allocator Design Questions

\[ \text{p1} = \text{malloc}(4) \]
\[ \text{p2} = \text{malloc}(5) \]
\[ \text{p3} = \text{malloc}(6) \]
\[ \text{free(p2)} \]
\[ \text{p4} = \text{malloc}(2) \]
Allocator Design Questions

p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)

1 How does free know an allocated block’s size?
Allocator Design Questions

\[ p_1 = \text{malloc}(4) \]
\[ p_2 = \text{malloc}(5) \]
\[ p_3 = \text{malloc}(6) \]
\[ \text{free}(p_2) \]
\[ p_4 = \text{malloc}(2) \]

2. How is unallocated space represented?
Allocator Design Questions

\[ p_1 = \text{malloc}(4) \]
\[ p_2 = \text{malloc}(5) \]
\[ p_3 = \text{malloc}(6) \]
\[ \text{free}(p_2) \]
\[ p_4 = \text{malloc}(2) \]

3 How is unallocated space selected for each allocation?
### Allocator Design Questions

| p1 = malloc(4)               |               |
| p2 = malloc(5)               |               |
| p3 = malloc(6)               |               |
| free(p2)                     |               |
| p4 = malloc(2)               |               |

4. How finely is unallocated space tracked?
Allocator Design Questions

\[
p1 = \text{malloc}(4) \quad \begin{array}{ccccccccc}
\hline
\hline
\hline
\hline
\hline
\hline
\hline
\hline
\end{array}
\]

\[
p2 = \text{malloc}(5) \quad \begin{array}{ccccccccc}
\hline
\hline
\hline
\hline
\hline
\hline
\hline
\hline
\hline
\end{array}
\]

\[
p3 = \text{malloc}(6) \quad \begin{array}{ccccccccc}
\hline
\hline
\hline
\hline
\hline
\hline
\hline
\hline
\hline
\hline
\hline
\end{array}
\]

\[
\text{free}(p2) \quad \begin{array}{ccccccccc}
\hline
\hline
\hline
\hline
\hline
\hline
\hline
\hline
\hline
\hline
\hline
\end{array}
\]

\[
p4 = \text{malloc}(2) \quad \begin{array}{ccccccccc}
\hline
\hline
\hline
\hline
\hline
\hline
\hline
\hline
\hline
\hline
\end{array}
\]

5 When are more pages needed from the kernel?
Allocator Design Questions

1. How does `free` know an allocated block’s size?

2. How is unallocated space represented?

3. How is unallocated space selected for each allocation?

4. How finely is unallocated space tracked?

5. When are more pages needed from the kernel?
Naive sbrk Allocator

\[
\begin{align*}
\text{p1} &= \text{malloc}(4) \\
\text{p2} &= \text{malloc}(5) \\
\text{p3} &= \text{malloc}(6) \\
\text{free(p2)} \\
\text{p4} &= \text{malloc}(2)
\end{align*}
\]
Naive `sbrk` Allocator

```
p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
p4 = malloc(2)
```

1️⃣ How does `free` know an allocated block’s size?

*It doesn’t*
p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
p4 = malloc(2)

2 How is unallocated space represented?

It isn’t
Naive *sbrk* Allocator

3. How is unallocated space selected for each allocation?

*Always add to the end*
Naive sbrk Allocator

\[ p_1 = \text{malloc}(4) \]  
\[ p_2 = \text{malloc}(5) \]  
\[ p_3 = \text{malloc}(6) \]  
\[ \text{free}(p_2) \]  
\[ p_4 = \text{malloc}(2) \]

\[ \text{4 How finely is unallocated space tracked?} \]  
\[ \text{Nothing to track} \]
### Naive `sbrk` Allocator

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>p1 = malloc(4)</code></td>
<td>![Diagram 1]</td>
</tr>
<tr>
<td><code>p2 = malloc(5)</code></td>
<td>![Diagram 2]</td>
</tr>
<tr>
<td><code>p3 = malloc(6)</code></td>
<td>![Diagram 3]</td>
</tr>
<tr>
<td><code>free(p2)</code></td>
<td>![Diagram 4]</td>
</tr>
<tr>
<td><code>p4 = malloc(2)</code></td>
<td>![Diagram 5]</td>
</tr>
</tbody>
</table>

5. When are more pages needed from the kernel?

*Every allocation*
Naive `sbrk` Allocator

Real allocator needs to produce pointers aligned on 16 bytes:

```c
#define ALIGNMENT 16
#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~(ALIGNMENT-1))

void *mm_malloc(size_t size) {
    return sbrk(ALIGN(size));
}

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

How finely is unallocated space tracked?

Some unallocated space can be left in a block for alignment padding
Picture Conventions

Since an implementation aligns to 16 bytes:

\[ = 16 \text{ bytes, a “word”} \]

\[
N = N \times 16 \text{ bytes}
\]

\[
\text{p1 = malloc(4) allocation of 64 bytes}
\]
Naive Chunked `sbrk` Allocator

Chunk size of 6:

\[ p_1 = \text{malloc}(4) \]
\[ p_2 = \text{malloc}(5) \]
\[ p_3 = \text{malloc}(6) \]
\[ \text{free}(p_2) \]
\[ p_4 = \text{malloc}(2) \]
Naive Chunked \texttt{sbrk} Allocator

Chunk size of 6:

\begin{align*}
\text{p1} &= \text{malloc}(4) \\
\text{p2} &= \text{malloc}(5) \\
\text{p3} &= \text{malloc}(6) \\
\text{free}(\text{p2}) \\
\text{p4} &= \text{malloc}(2)
\end{align*}

When are more pages needed from the kernel?

When more is needed for an allocation
Naive Chunked `sbrk` Allocator

Pick a chunk size:

#define CHUNK_SIZE (1 << 14)
#define CHUNK_ALIGN(size) (((size)+(CHUNK_SIZE-1)) & ~(CHUNK_SIZE-1))
Naive Chunked \texttt{sbrk} Allocator

```c
void *current_avail = NULL;
size_t current_avail_size = 0;

int mm_init() {
    current_avail = sbrk(0);
    current_avail_size = 0;
    return 0;
}
```
Naive Chunked \textit{sbrk} Allocator

```c
void *mm_malloc(size_t size) {
    size_t newsize = ALIGN(size);
    void *p;

    if (current_avail_size < newsize) {
        sbrk(CHUNK_ALIGN(newsize));
        current_avail_size += CHUNK_ALIGN(newsize);
    }

    p = current_avail;
    current_avail += newsize;
    current_avail_size -= newsize;

    return p;
}
```
Naive mmap Allocator

Page size of 8:

\[ p1 = malloc(4) \]
\[ p2 = malloc(5) \]
\[ p3 = malloc(6) \]
\[ \text{free}(p2) \]
\[ p4 = malloc(2) \]
Naive mmap Allocator

When are more pages needed from the kernel?

When the most recent page doesn’t have space

Page size of 8:

- \( p_1 = \text{malloc}(4) \)
- \( p_2 = \text{malloc}(5) \)
- \( p_3 = \text{malloc}(6) \)
- \( \text{free}(p_2) \)
- \( p_4 = \text{malloc}(2) \)
Naive \texttt{mmap} Allocator

```c
void *current_avail = NULL;
size_t current_avail_size = 0;

void *mm_malloc(size_t size) {
    size_t newsize = ALIGN(size);
    void *p;

    if (current_avail_size < newsize) {
        current_avail = mmap(0, CHUNK_ALIGN(newsize),
                              PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANON,
                              -1, 0);
        current_avail_size = CHUNK_ALIGN(newsize);
    }

    p = current_avail;
    current_avail += newsize;
    current_avail_size -= newsize;

    return p;
}
```
Fragmentation

Unallocated space in mapped pages is wasted

Naive `sbrk`:

Naive `mmap`:
Fragmentation

Unallocated space in mapped pages is wasted

Naive `sbrk`:

Naive `mmap`:

wasted space = \textit{fragmentation}
Fragmentation

Unallocated space in mapped pages is wasted

Naive `sbrk`:

```
```

Naive `mmap`:

```
```

Pick page chunk $\gg$ allocation size
Fragmentation

Unallocated space in mapped pages is wasted

Naive `sbrk`:

Naive `mmap`:
Fragmentation

Unallocated space in mapped pages is wasted

Naive `sbrk`:

![Diagram of naive sbrk]

Naive `mmap`:

![Diagram of naive mmap]

Taking `free` into account, both naive implementations suffer from extreme fragmentation

... so we need to keep track of unallocated space
Allocation Bit in a Block Header

\[ p_1 = \text{malloc}(4) \]
\[ p_2 = \text{malloc}(5) \]
\[ p_3 = \text{malloc}(6) \]
\[ \text{free}(p_2) \]
\[ p_4 = \text{malloc}(2) \]
Allocation Bit in a Block Header

```
p1 = malloc(4)  # 1100
p2 = malloc(5)  # 1110
p3 = malloc(6)  # 1111
free(p2)     # 1011
p4 = malloc(2)  # 1101
```
Size + Allocation Bit in a Block Header

```c
p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
p4 = malloc(2)
```
Sizes in a Block Header ⇒ Implicit Free List

\[
p_1 = \text{malloc(4)}
\]

\[
p_2 = \text{malloc(5)}
\]

\[
p_3 = \text{malloc(6)}
\]

\[
\text{free}(p_2)
\]

\[
p_4 = \text{malloc(2)}
\]
Sizes in a Block Header ⇒ Implicit Free List

\[
\begin{align*}
\text{p1} &= \text{malloc}(4) \\
\text{p2} &= \text{malloc}(5) \\
\text{p3} &= \text{malloc}(6) \\
\text{free(p2)} \\
\text{p4} &= \text{malloc}(2)
\end{align*}
\]

How does \text{free} know an allocated block's size?

\text{It's stored at the start of the block}
Sizes in a Block Header ⇒ Implicit Free List

\( p1 = \text{malloc}(4) \)

\( p2 = \text{malloc}(5) \)

\( p3 = \text{malloc}(6) \)

\( \text{free}(p2) \)

\( p4 = \text{malloc}(2) \)

How is unallocated space represented?

A bit in the block header distinguishes allocated from unallocated
Sizes in a Block Header ⇒ Implicit Free List

p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
p4 = malloc(2)

3 How is unallocated space selected for each allocation?

Search the chain for a block that’s free and big enough
Sizes in a Block Header ⇒ Implicit Free List

p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)

4 How finely is unallocated space tracked?

A block should be of size 2 or more to be useful
Sizes in a Block Header ⇒ Implicit Free List

- **p1 = malloc(4)**
- **p2 = malloc(5)**
- **p3 = malloc(6)**
- **free(p2)**
- **p4 = malloc(2)**

5 When are more pages needed from the kernel?

*When a search through the chain doesn’t find a free block that’s big enough*
Terminating the Block List

How does the allocator know that the size-2 block is the last one?

Compare the next pointer to an end-of-heap address

or

Add a “zero”-sized block to terminate the chain
Storing the Size and Allocation Bit

typedef struct {
    size_t size;
    char   allocated;
} block_header;
Storing the Size and Allocation Bit

typedef struct {
    size_t size;
    char   allocated;
} block_header;

Copy
Storing the Size and Allocation Bit

```c
typedef struct {
    size_t size;
    char allocated;
} block_header;
```
Storing the Size and Allocation Bit

```c
typedef struct {
    size_t size;
    char   allocated;
} block_header;
```

```
sizeof(block_header) = 16
```

Aligned payload size ⇒ 16-byte alignment preserved

... although that’s a lot of empty space
Storing the Size and Allocation Bit

```
typedef struct {
    size_t size;
    char    allocated;
} block_header;
```

Macro for block overhead:

```
#define OVERHEAD sizeof(block_header)
```
Storing the Size and Allocation Bit

```
typedef struct {
    size_t size;
    char   allocated;
} block_header;
```

Macro for getting the header from a payload pointer:

```
#define HDRP(bp) (((char *)(bp) - sizeof(block_header))
```
Storing the Size and Allocation Bit

```
typedef struct {
    size_t size;
    char   allocated;
} block_header;
```

Macros for working with a raw pointer as the header:

```
#define GET_SIZE(p)  ((block_header *)(p))->size
#define GET_ALLOC(p) ((block_header *)(p))->allocated
```
Storing the Size and Allocation Bit

```c
typedef struct {
    size_t size;
    char     allocated;
} block_header;
```

Macro for getting the next block’s payload:

```c
#define NEXT_BLKP(bp) (((char *)(bp) + GET_SIZE(HDRP(bp)))
```
Initializing the Allocator

```c
void *first_bp;

int mm_init() {
    sbrk(sizeof(block_header));
    first_bp = sbrk(0);

    GET_SIZE(HDRP(first_bp)) = 0;
    GET_ALLOC(HDRP(first_bp)) = 1;

    return 0;
}
```
Initializing the Allocator

```c
void *first_bp;

int mm_init() {
    sbrk(sizeof(block_header));
    first_bp = sbrk(0);

    GET_SIZE(HDRP(first_bp)) = 0;
    GET_ALLOC(HDRP(first_bp)) = 1;

    return 0;
}
```
void *first_bp;

int mm_init() {
  sbrk(sizeof(block_header));
  first_bp = sbrk(0);

  GET_SIZE(HDRP(first_bp)) = 0;
  GET_ALLOC(HDRP(first_bp)) = 1;

  return 0;
}
Initializing the Allocator

```c
void *first_bp;

int mm_init() {  
    sbrk(sizeof(block_header));  
    first_bp = sbrk(0);

    GET_SIZE(HDRP(first_bp)) = 0;
    GET_ALLOC(HDRP(first_bp)) = 1;

    return 0;
}
```
void extend(size_t new_size) {
    size_t chunk_size = CHUNK_ALIGN(new_size);
    void *bp = sbrk(chunk_size);

    GET_SIZE(HDRP(bp)) = chunk_size;
    GET_ALLOC(HDRP(bp)) = 0;

    GET_SIZE(HDRP(NEXT_BLKBP(bp))) = 0;
    GET_ALLOC(HDRP(NEXT_BLKBP(bp))) = 1;
}
Adding Pages

```c
void extend(size_t new_size) {
    size_t chunk_size = CHUNK_ALIGN(new_size);
    void *bp = sbrk(chunk_size);

    GET_SIZE(HDRP(bp)) = chunk_size;
    GET_ALLOC(HDRP(bp)) = 0;

    GET_SIZE(HDRP(NEXT_BLKP(bp))) = 0;
    GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 1;
}
```
Adding Pages

```c
void extend(size_t new_size) {
    size_t chunk_size = CHUNK_ALIGN(new_size);
    void *bp = sbrk(chunk_size);

    GET_SIZE(HDRP(bp)) = chunk_size;
    GET_ALLOC(HDRP(bp)) = 0;

    GET_SIZE(HDRP(NEXT_BLKP(bp))) = 0;
    GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 1;
}
```
Adding Pages

```c
void extend(size_t new_size) {
    size_t chunk_size = CHUNK_ALIGN(new_size);
    void *bp = sbrk(chunk_size);

    GET_SIZE(HDRP(bp)) = chunk_size;
    GET_ALLOC(HDRP(bp)) = 0;

    GET_SIZE(HDRP(NEXT_BLKP(bp))) = 0;
    GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 1;
}
```
Adding Pages

```c
void extend(size_t new_size) {
    size_t chunk_size = CHUNK_ALIGN(new_size);
    void *bp = sbrk(chunk_size);

    GET_SIZE(HDRP(bp)) = chunk_size;
    GET_ALLOC(HDRP(bp)) = 0;

    GET_SIZE(HDRP(NEXT_BLKP(bp))) = 0;
    GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 1;
}
```
Adding Pages

```c
void extend(size_t new_size) {
    size_t chunk_size = CHUNK_ALIGN(new_size);
    void *bp = sbrk(chunk_size);

    GET_SIZE(HDRP(bp)) = chunk_size;
    GET_ALLOC(HDRP(bp)) = 0;

    GET_SIZE(HDRP(NEXT_BLKP(bp))) = 0;
    GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 1;
}
```
Adding Pages

```c
void extend(size_t new_size) {
    size_t chunk_size = CHUNK_ALIGN(new_size);
    void *bp = sbrk(chunk_size);

    GET_SIZE(HDRP(bp)) = chunk_size;
    GET_ALLOC(HDRP(bp)) = 0;

    GET_SIZE(HDRP(NEXT_BLKP(bp))) = 0;
    GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 1;
}
```
Adding Pages

```c
void extend(size_t new_size) {
    size_t chunk_size = CHUNK_ALIGN(new_size);
    void *bp = sbrk(chunk_size);

    GET_SIZE(HDRP(bp)) = chunk_size;
    GET_ALLOC(HDRP(bp)) = 0;

    GET_SIZE(HDRP(NEXT_BLKP(bp))) = 0;
    GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 1;
}
```
Adding Pages

```c
void extend(size_t new_size) {
    size_t chunk_size = CHUNK_ALIGN(new_size);
    void *bp = sbrk(chunk_size);

    GET_SIZE(HDRP(bp)) = chunk_size;
    GET_ALLOC(HDRP(bp)) = 0;

    GET_SIZE(HDRP(NEXT_BLKP(bp))) = 0;
    GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 1;
}
```
Finding a Block to Allocate

```c
void *mm_malloc(size_t size) {
    int new_size = ALIGN(size + OVERHEAD);
    void *bp = first_bp;

    while (GET_SIZE(HDRP(bp)) != 0) {
        if (!GET_ALLOC(HDRP(bp))
            && (GET_SIZE(HDRP(bp)) >= new_size)) {
            set_allocated(bp, new_size);
            return bp;
        }
        bp = NEXT_BLKP(bp);
    }

    extend(new_size);
    set_allocated(bp, new_size);
    return bp;
}
```
void *mm_malloc(size_t size) {
    int new_size = ALIGN(size + OVERHEAD);
    void *bp = first_bp;

    while (GET_SIZE(HDRP(bp)) != 0) {
        if (!GET_ALLOC(HDRP(bp))
            && (GET_SIZE(HDRP(bp)) >= new_size)) {
            set_allocated(bp, new_size);
            return bp;
        }
        bp = NEXT_BLKP(bp);
    }
    extend(new_size);
    set_allocated(bp, new_size);
    return bp;
}
void set_allocated(void *bp, size_t size) {
    size_t extra_size = GET_SIZE(HDRP(bp)) - size;

    if (extra_size > ALIGN(1 + OVERHEAD)) {
        GET_SIZE(HDRP(bp)) = size;
        GET_SIZE(HDRP(NEXT_BLKP(bp))) = extra_size;
        GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 0;
    }

    GET_ALLOC(HDRP(bp)) = 1;
}
void set_allocated(void *bp, size_t size) {
    size_t extra_size = GET_SIZE(HDRP(bp)) - size;

    if (extra_size > ALIGN(1 + OVERHEAD)) {
        GET_SIZE(HDRP(bp)) = size;
        GET_SIZE(HDRP(NEXT_BLKP(bp))) = extra_size;
        GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 0;
    }

    GET_ALLOC(HDRP(bp)) = 1;
}
Marking a Block as Allocated

```c
void set_allocated(void *bp, size_t size) {
    size_t extra_size = GET_SIZE(HDRP(bp)) - size;

    if (extra_size > ALIGN(1 + OVERHEAD)) {
        GET_SIZE(HDRP(bp)) = size;
        GET_SIZE(HDRP(NEXT_BLKP(bp))) = extra_size;
        GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 0;
    }

    GET_ALLOC(HDRP(bp)) = 1;
}
```
void set_allocated(void *bp, size_t size) {
    size_t extra_size = GET_SIZE(HDRP(bp)) - size;

    if (extra_size > ALIGN(1 + OVERHEAD)) {
        GET_SIZE(HDRP(bp)) = size;
        GET_SIZE(HDRP(NEXT_BLKP(bp))) = extra_size;
        GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 0;
    }

    GET_ALLOC(HDRP(bp)) = 1;
}
Marking a Block as Allocated

```c
void set_allocated(void *bp, size_t size) {
    size_t extra_size = GET_SIZE(HDRP(bp)) - size;

    if (extra_size > ALIGN(1 + OVERHEAD)) {
        GET_SIZE(HDRP(bp)) = size;
        GET_SIZE(HDRP(NEXT_BLKP(bp))) = extra_size;
        GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 0;
    }

    GET_ALLOC(HDRP(bp)) = 1;
}
```
void set_allocated(void *bp, size_t size) {
    size_t extra_size = GET_SIZE(HDRP(bp)) - size;

    if (extra_size > ALIGN(1 + OVERHEAD)) {
        GET_SIZE(HDRP(bp)) = size;
        GET_SIZE(HDRP(NEXT_BLKP(bp))) = extra_size;
        GET_ALLOCS(HDRP(NEXT_BLKP(bp))) = 0;
    }

    GET_ALLOCS(HDRP(bp)) = 1;
}
Freeing a Block

```c
void mm_free(void *bp) {
    GET_ALLOC(HDRP(bp)) = 0;
}
```
Freeing a Block

```c
void mm_free(void *bp) {
    GET_ALLOC(HDRP(bp)) = 0;
}
```
Freeing Multiple Blocks

free(p2)

malloc(5)

free(p3)
Freeing Multiple Blocks

\text{free}(p2)

\begin{align*}
\text{malloc}(5) & \quad \Rightarrow \\
& \quad \text{there's room here, but no unallocated block is big enough} \Rightarrow \text{extra fragmentation}
\end{align*}
Freeing Multiple Blocks

```c
free(p2)
```

```c
malloc(5)
```

**free** should **coalesce** adjacent unallocated blocks
Coalescing Unallocated Blocks

Needed invariant: no two unallocated blocks are adjacent
can maintain at each \texttt{free} call

For \texttt{free(p2)}:

- No merge
- Merge with next block
- Merge with previous block
- Merge with both blocks
Coalescing Unallocated Blocks

Needed invariant: no two unallocated blocks are adjacent
\[ \text{can maintain at each } \texttt{free} \text{ call} \]

For \( \texttt{free(p2)} \):

\[ \cdots \begin{array}{c} 3 \\ 1 \end{array} \begin{array}{c} 4 \\ 1 \end{array} \begin{array}{c} 3 \\ 1 \end{array} \cdots \] no merge

\[ \cdots \begin{array}{c} 3 \\ 1 \end{array} \begin{array}{c} 4 \\ 0 \end{array} \begin{array}{c} 3 \\ 1 \end{array} \cdots \]

\[ \cdots \begin{array}{c} 3 \\ 1 \end{array} \begin{array}{c} 4 \\ 1 \end{array} \begin{array}{c} 3 \\ 0 \end{array} \cdots \] merge with previous block

\[ \begin{array}{c} 3 \\ 0 \end{array} \begin{array}{c} 4 \\ 1 \end{array} \begin{array}{c} 3 \\ 1 \end{array} \cdots \]

\[ \begin{array}{c} 7 \\ 0 \end{array} \begin{array}{c} 3 \\ 1 \end{array} \cdots \] merge with both blocks

Need to find the block before \( \texttt{p2} \)
Blocks with Headers and Footers
typedef struct {
    size_t size;
    int filler;
} block_footer;
Blocks with Headers and Footers

typedef struct {
    size_t size;
    int filler;
} block_footer;
Blocks with Headers and Footers

typedef struct {
    size_t size;
    int filler;
} block_footer;

#define OVERHEAD (sizeof(block_header)+sizeof(block_footer))
Blocks with Headers and Footers

```
typedef struct {
    size_t size;
    int filler;
} block_footer;
```

```
#define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE((char *)(bp) - OVERHEAD))
```
Blocks with Headers and Footers

typedef struct {
    size_t size;
    int filler;
} block_footer;

#define FTRP(bp) ((char *)(bp)+GET_SIZE(HDRP(bp))-OVERHEAD)
void extend(size_t new_size) {
    ....
    GET_SIZE(HDRP(bp)) = chunk_size;
    GET_SIZE(FTRP(bp)) = chunk_size;
    ....
}

void set_allocated(void *bp, size_t size) {
    ....
    GET_SIZE(HDRP(bp)) = size;
    GET_SIZE(FTRP(bp)) = size;
    GET_SIZE(HDRP(NEXT_BLKP(bp))) = extra_size;
    GET_SIZE(FTRP(NEXT_BLKP(bp))) = extra_size;
    ....
}
Coalescing after Free

```c
void mm_free(void *bp) {
    GET_ALLOC(HDRP(bp)) = 0;
    coalesce(bp);
}
```
Coalescing Free Blocks

```c
void *coalesce(void *bp) {
    size_t prev_alloc = GET_ALLOC(HDRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));
    ....

    return bp;
}
```
Coalescing Free Blocks

```c
void *coalesce(void *bp) {
    size_t prev_alloc = GET_ALLOC(HDRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));

    if (prev_alloc && next_alloc) {           /* Case 1 */
        /* nothing to do */
    }
    /* nothing to do */
    ....
}
```
Coalescing Free Blocks

```c
void *coalesce(void *bp) {
    size_t prev_alloc = GET_ALLOC(HDRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));
    ....

    else if (prev_alloc && !next_alloc) { /* Case 2 */
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        GET_SIZE(HDRP(bp)) = size;
        GET_SIZE(FTRP(bp)) = size;
    }
    ....
}
```

**Diagram:**
- `bp` is the current block pointer.
- The illustration shows how adjacent free blocks can be coalesced.
- The coalescing process merges adjacent blocks with the same size.

**Example:**
- Before coalescing: `bp` points to blocks with sizes `[4, 5, 5, 4]`.
- After coalescing: `bp` points to blocks with sizes `[4, 9]`.

**Copy:**
- The code snippet is highlighted for emphasis.
void *coalesce(void *bp) {
    size_t prev_alloc = GET_ALLOC(HDRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));
    ....

    else if (!prev_alloc && next_alloc) {  /* Case 3 */
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        GET_SIZE(FTRP(bp)) = size;
        GET_SIZE(HDRP(PREV_BLKP(bp))) = size;
        bp = PREV_BLKP(bp);
    }
    ....
}
void *coalesce(void *bp) {
    size_t prev_alloc = GET_ALLOC(HDRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));
    ....

    else { /* Case 4 */
        size += (GET_SIZE(HDRP(PREV_BLKP(bp)))
            + GET_SIZE(HDRP(NEXT_BLKP(bp))));
        GET_SIZE(HDRP(PREV_BLKP(bp))) = size;
        GET_SIZE(FTRP(NEXT_BLKP(bp))) = size;
        bp = PREV_BLKP(bp);
    }
    ....
}
Prolog Block

Create a prolog block so **coalesce** can always look backwards

```c
int mm_init() {
    ....
    mm_malloc(0); /* never freed */
    ....
}
```