

# Malloc Lab - Part 2, Processes

# OMET Survey

- Please take some time to complete the OMET survey.

# Malloc Lab (Part 2)

```

struct block
{
    /* Header contains:
    * a. size
    * b. allocation flag
    */
    word_t header;

    union
    {
        struct
        {
            block_t *prev;
            block_t *next;
        } links;
        /*
        * We don't know what the size of the payload will be, so we will
        * declare it as a zero-length array. This allows us to obtain a
        * pointer to the start of the payload.
        */
        unsigned char data[0];

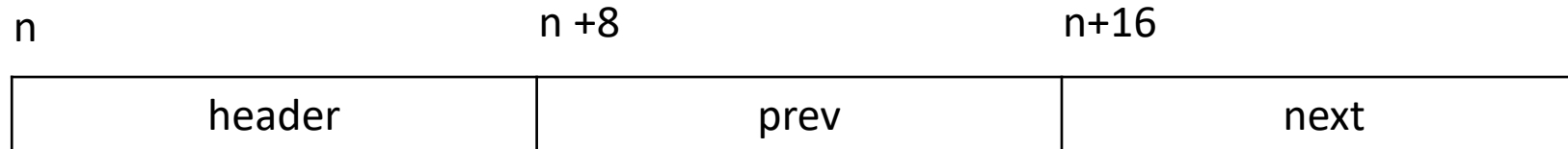
        /*
        * Payload contains:
        * a. only data if allocated
        * b. pointers to next/previous free blocks if unallocated
        */
    } payload;

    /*
    * We can't declare the footer as part of the struct, since its starting
    * position is unknown
    */
};

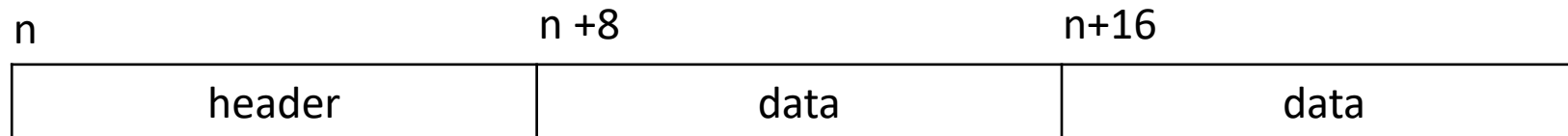
```

# struct block

If NOT allocated



If allocated



- You DO NOT need to put any data in the allocated space
  - **You just need to know the STARTING ADDRESS OF data (which is the address of header + size of a word)**
- Utilize the prev and next pointer when using explicit list traversal

# Let's See Some Code

```
int mm_init(void)
{
    /* Create the initial empty heap */
    word_t *start = (word_t *) (mem_sbrk(2*wsize));
    if ((ssize_t) start == -1) {
        printf("ERROR: mem_sbrk failed in mm_init, returning %p\n", start);
        return -1;
    }

    /* Prologue footer */
    start[0] = pack(0, true);
    /* Epilogue header */
    start[1] = pack(0, true);

    /* Heap starts with first "block header", currently the epilogue header */
    heap_start = (block_t *) &(start[1]);

    /* Extend the empty heap with a free block of chunksize bytes */
    block_t *free_block = extend_heap(chunksize);
    if (free_block == NULL) {
        printf("ERROR: extend_heap failed in mm_init, returning");
        return -1;
    }

    /* Set the head of the free list to this new free block */
    free_list_head = free_block;
    free_list_head->payload.links.prev = NULL;
    free_list_head->payload.links.next = NULL;

    return 0;
}
```

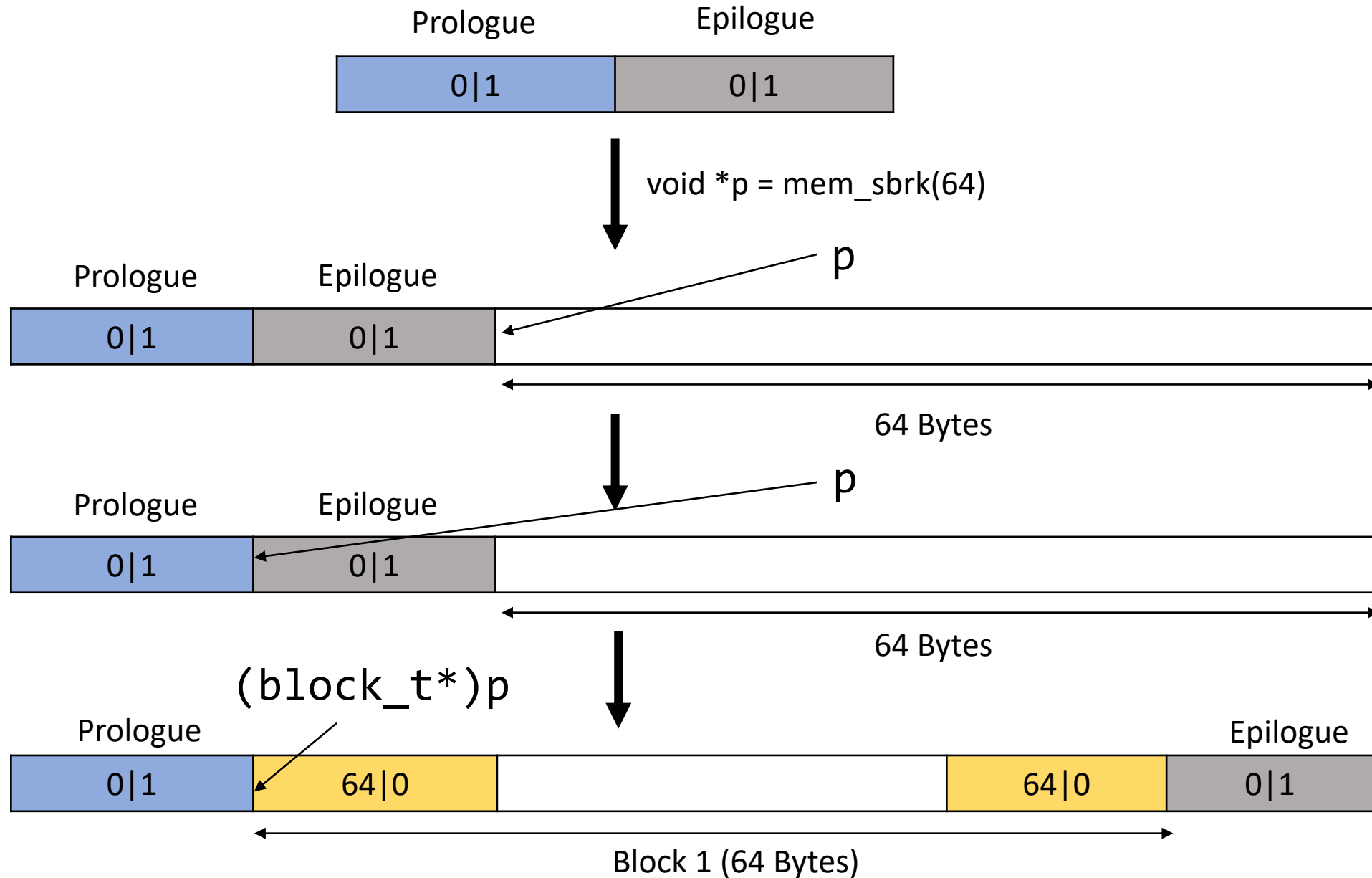
```
static block_t *extend_heap(size_t size)
{
    void *bp;

    // Allocate an even number of words to maintain alignment
    size = round_up(size, dsize);
    if ((bp = mem_sbrk(size)) == (void *) -1) {
        return NULL;
    }

    // bp is a pointer to the new memory block requested

    // TODO: Implement extend_heap.
    // You will want to replace this return statement...
    return NULL;
}
```

# Example: extend\_heap by 64 Bytes



# Let's See Some Code

```
int mm_init(void)
{
    /* Create the initial empty heap */
    word_t *start = (word_t *) (mem_sbrk(2*wsize));
    if ((ssize_t)start == -1) {
        printf("ERROR: mem_sbrk failed in mm_init, returning %p\n", start);
        return -1;
    }

    /* Prologue footer */
    start[0] = pack(0, true);
    /* Epilogue header */
    start[1] = pack(0, true);

    /* Heap starts with first "block header", currently the epilogue header */
    heap_start = (block_t *) &(start[1]);

    /* Extend the empty heap with a free block of chunksize bytes */
    block_t *free_block = extend_heap(chunksize);
    if (free_block == NULL) {
        printf("ERROR: extend_heap failed in mm_init, returning");
        return -1;
    }

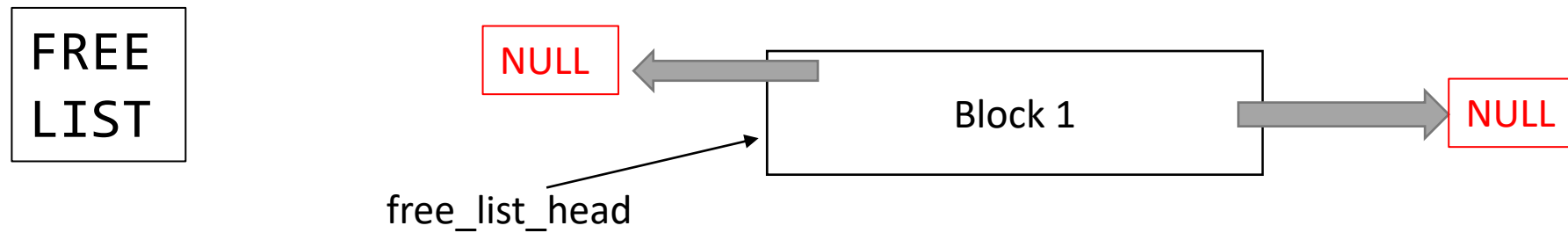
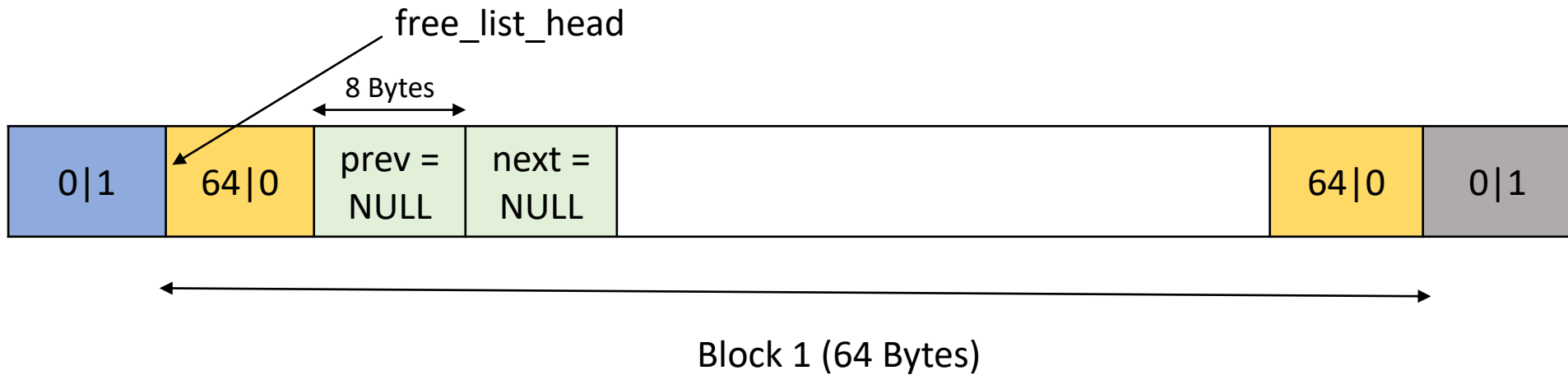
    /* Set the head of the free list to this new free block */
    free_list_head = free_block;
    free_list_head->payload.links.prev = NULL;
    free_list_head->payload.links.next = NULL;

    return 0;
}
```

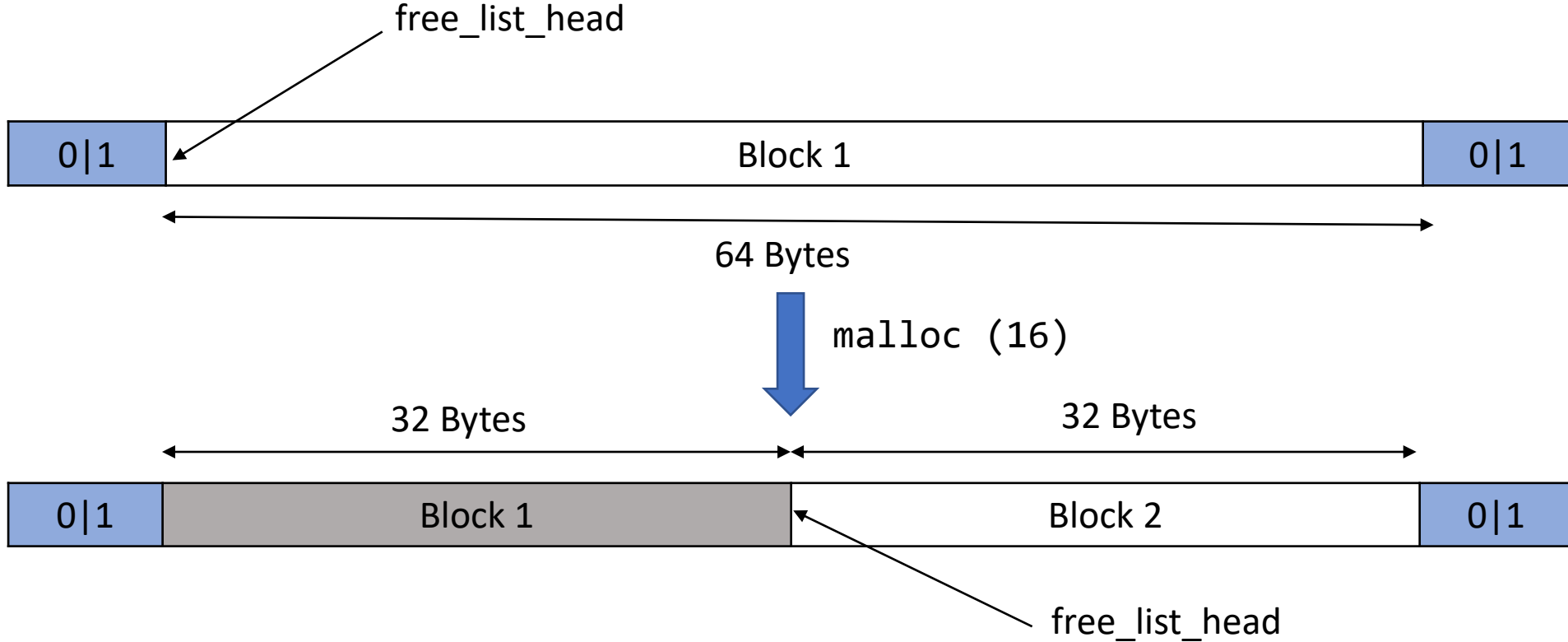
```
// Pointer to the first block in the free list
static block_t *free_list_head = NULL;
```



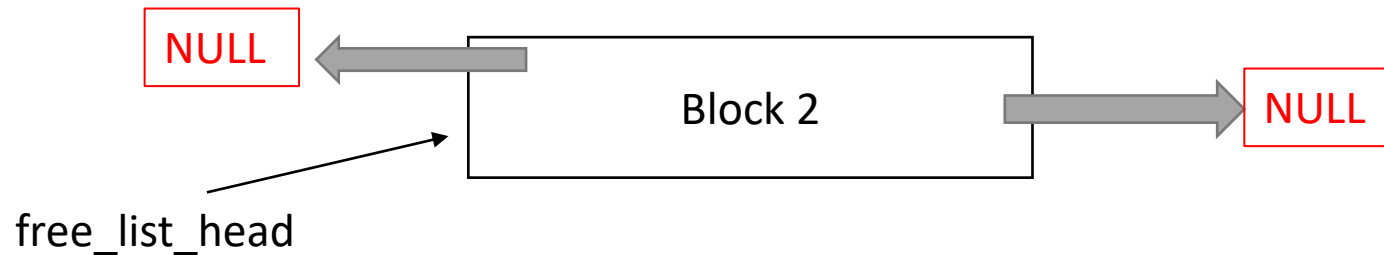
# Visualizing the Heap



# Allocating in the Heap



FREE  
LIST



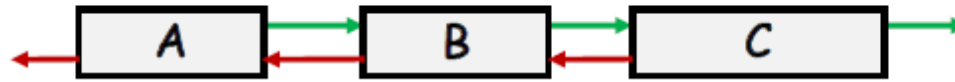
- Step 1: Remove Block 1 from Free List
- Step 2: Add Block 2 to Free List

# Free List

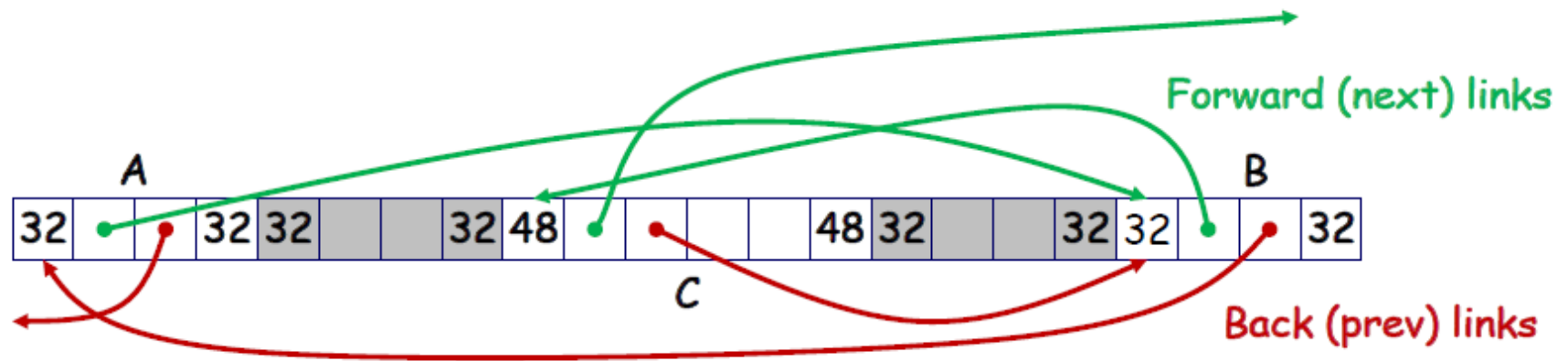
- The Free List is conceptually a doubly Linked List.
- A node (essentially a free block) in the Free list
  - Has pointer (prev) pointing to the previous free block
  - Has pointer (next) pointing to the next free block

# Explicit Free Lists

- Logically:

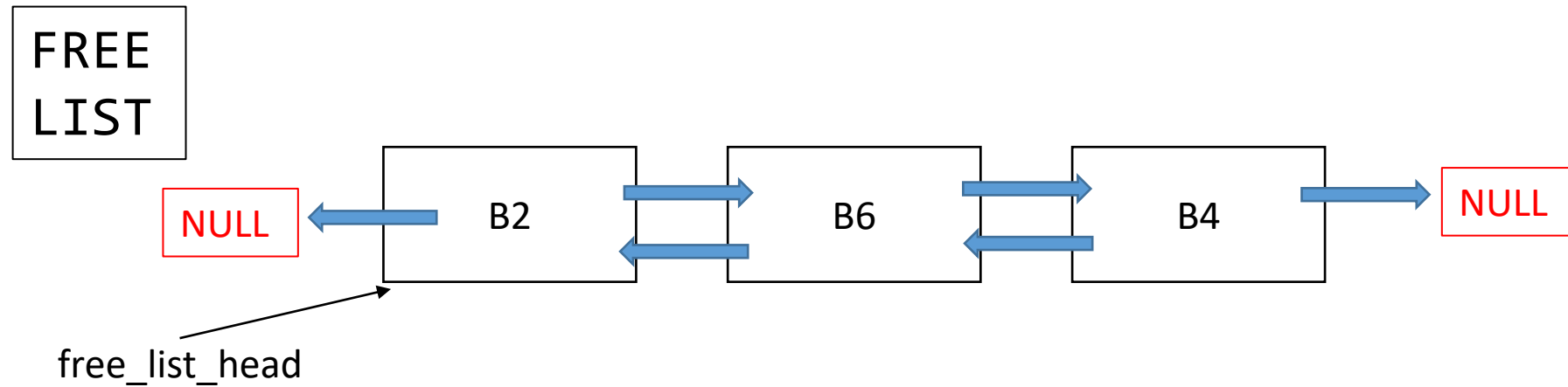
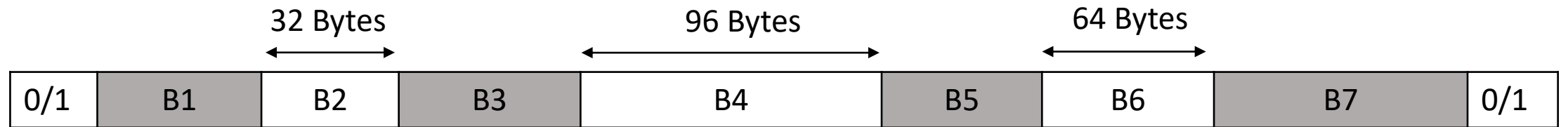


- Physically: blocks can be in any order



# Updating Free List During Allocation

# Consider this Scenario

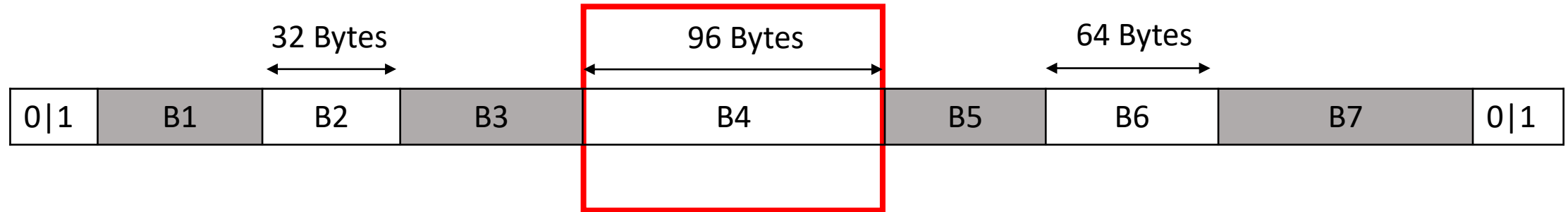


# CASE 1 (with Splitting)

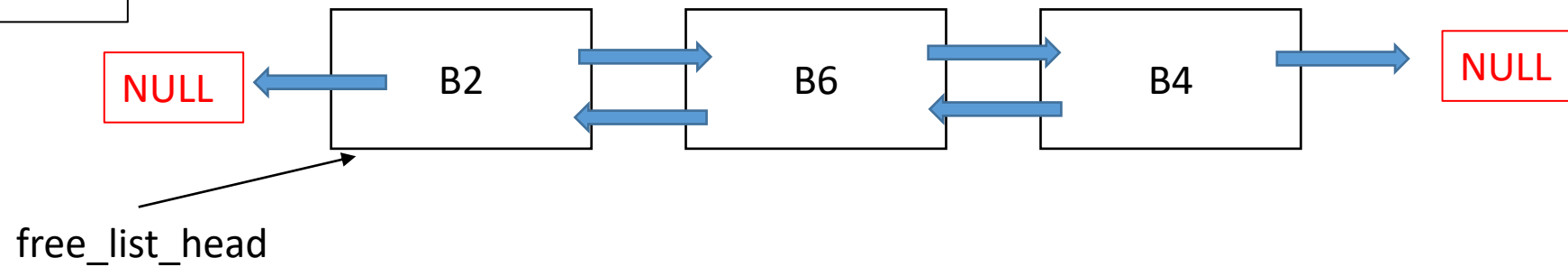
Which free block will you allocate in??

malloc(32)

Will depend on the Allocation Policy. Let's assume **Worst Fit** for this example



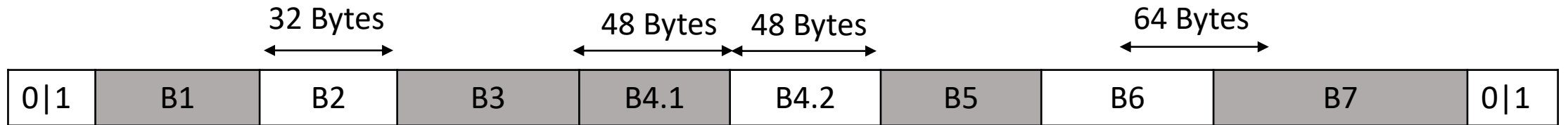
FREE  
LIST





malloc(32)

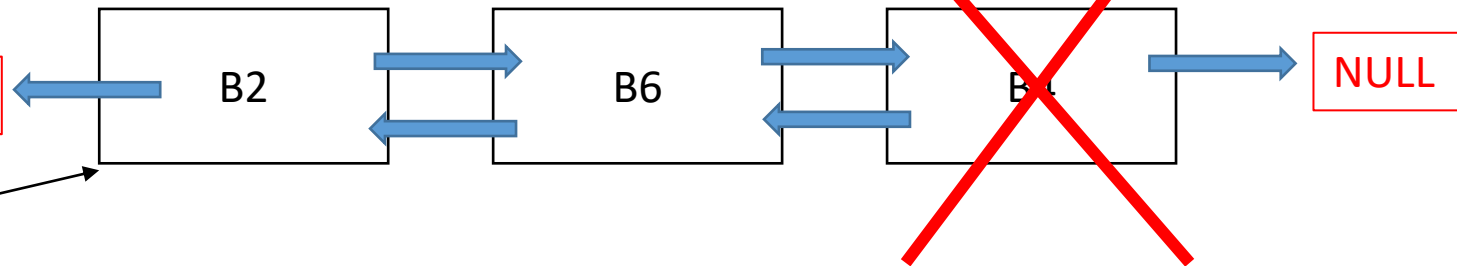
Splitting!!



FREE  
LIST

NULL

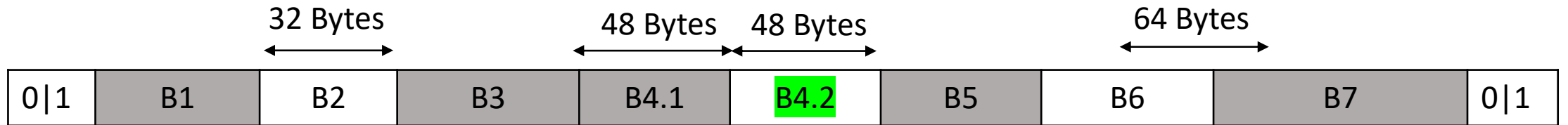
free\_list\_head



**Step 1:** Remove Block 4  
from FREE LIST

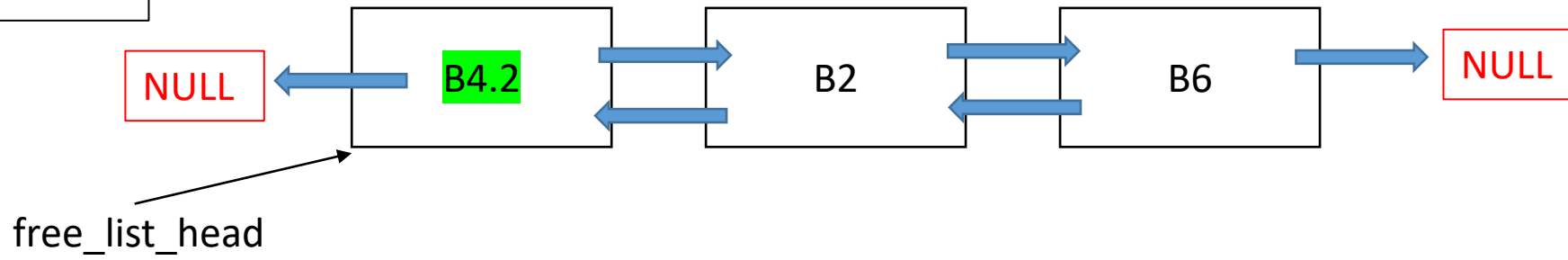
malloc(32)

Splitting!!



FREE  
LIST

**Step 2:** Insert Block 4.2  
to FREE LIST using LIFO

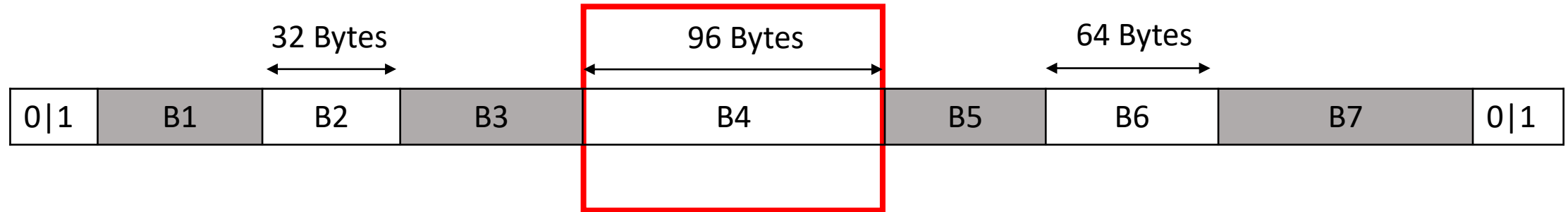


## CASE 2 (without Splitting)

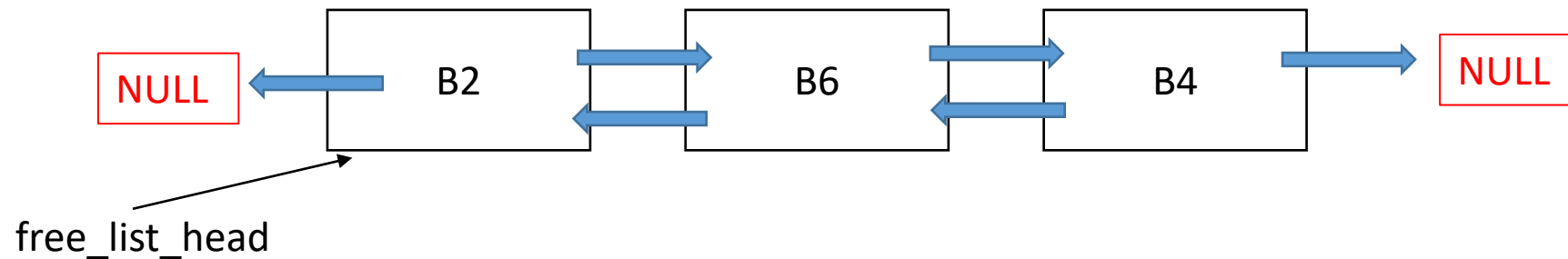
Which free block will you allocate in??

malloc(80)

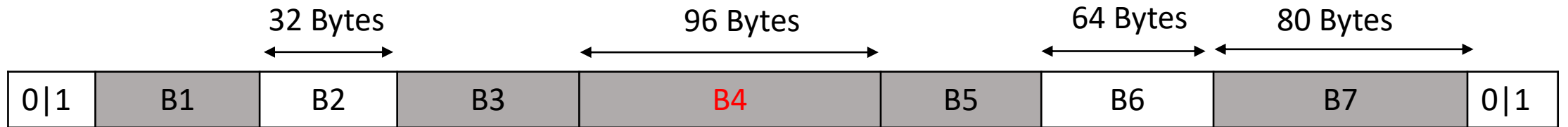
Will depend on the Allocation Policy. Let's assume **First Fit** for this example



FREE  
LIST

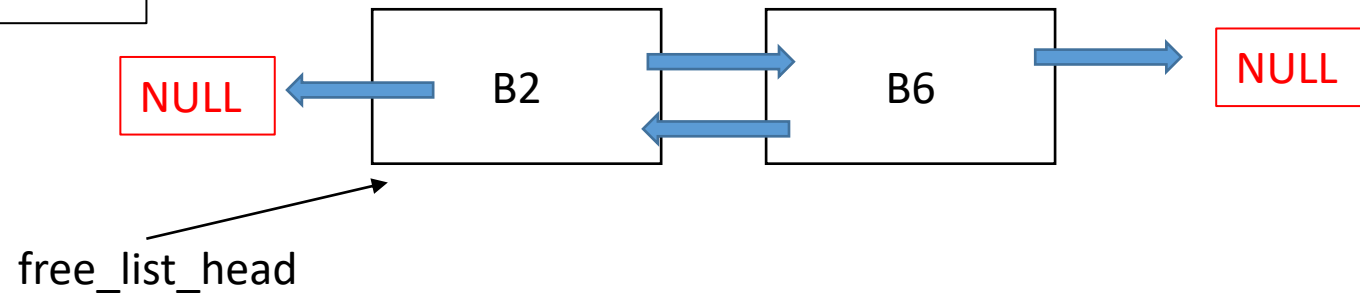


`malloc(80)`

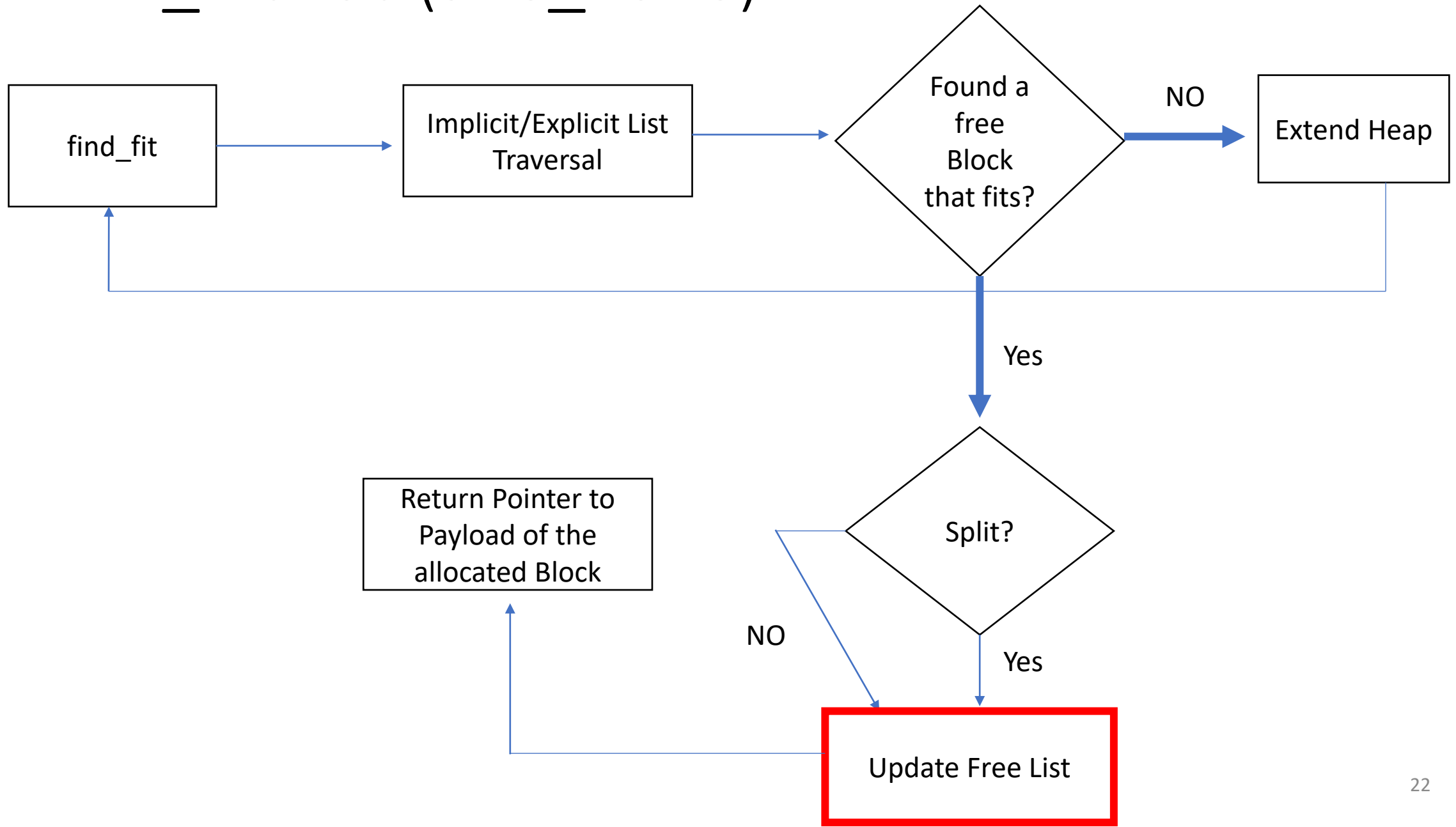


FREE  
LIST

Remove Block 4 from  
FREE LIST

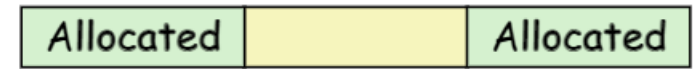


# mm\_malloc (size\_t size)

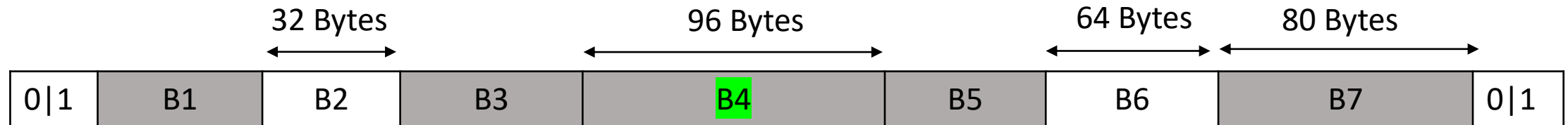


# Updating Free List During De-Allocation

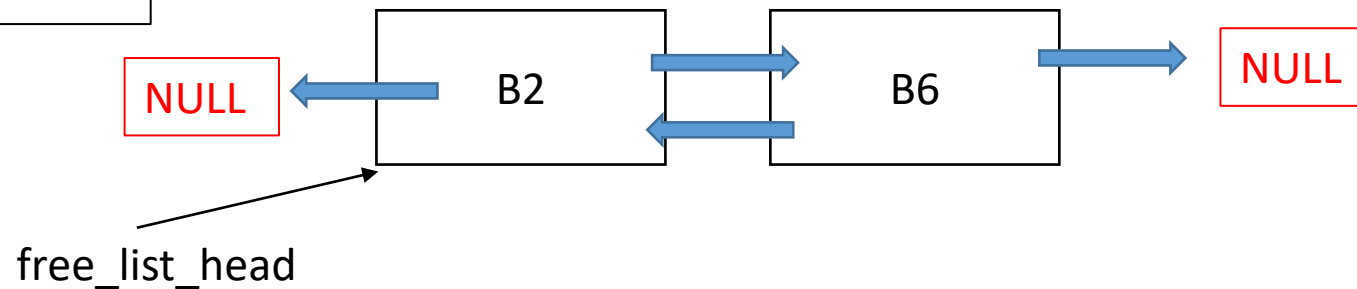
# Freeing A Block (Case 1)



Free Block 4

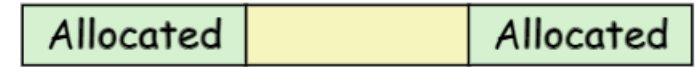


FREE  
LIST

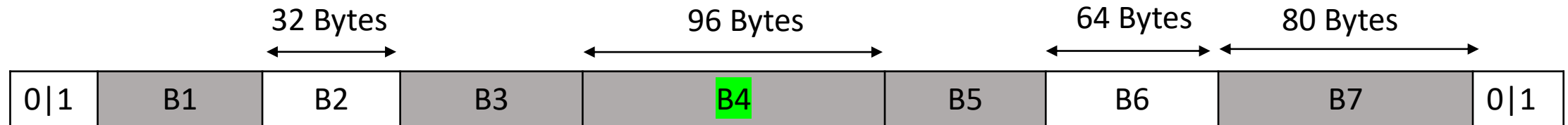




# Freeing A Block (Case 1)



Free Block 4



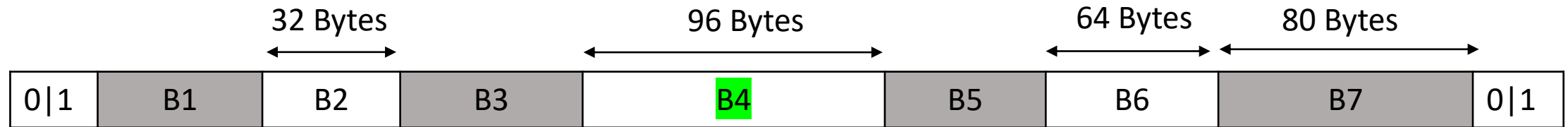
Where do you put Block 4 in the Free List?

- LIFO
- FIFO
- Address Ordered Policy

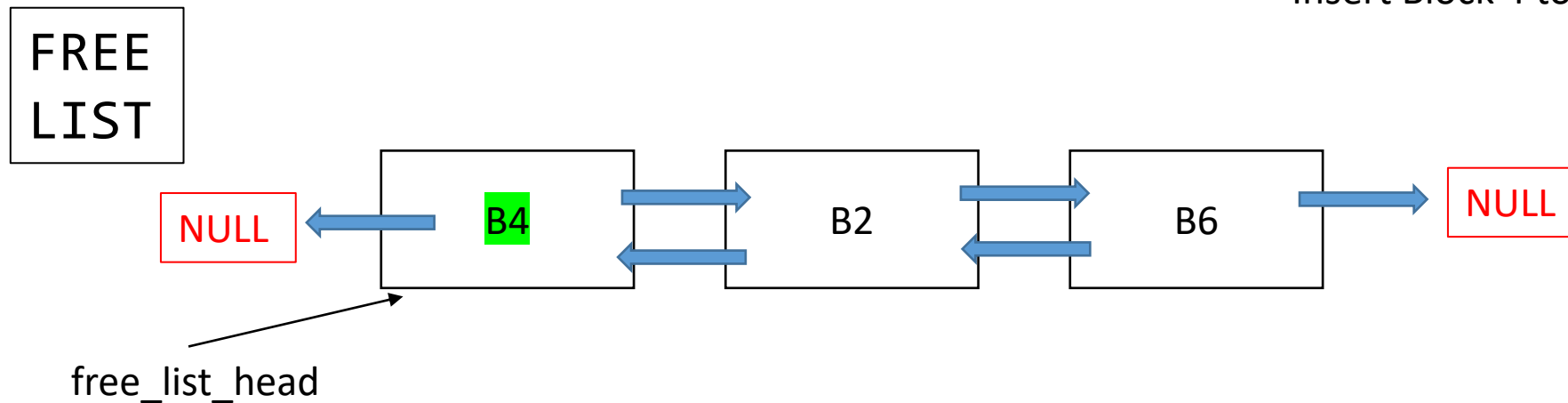
# Freeing A Block (Case 1)



Free Block 4



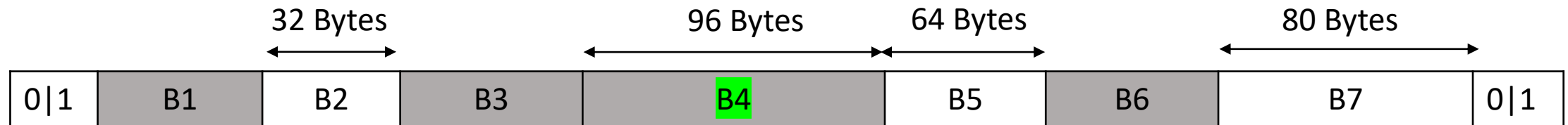
Insert Block 4 to the Free List



# Freeing A Block (Case 2)

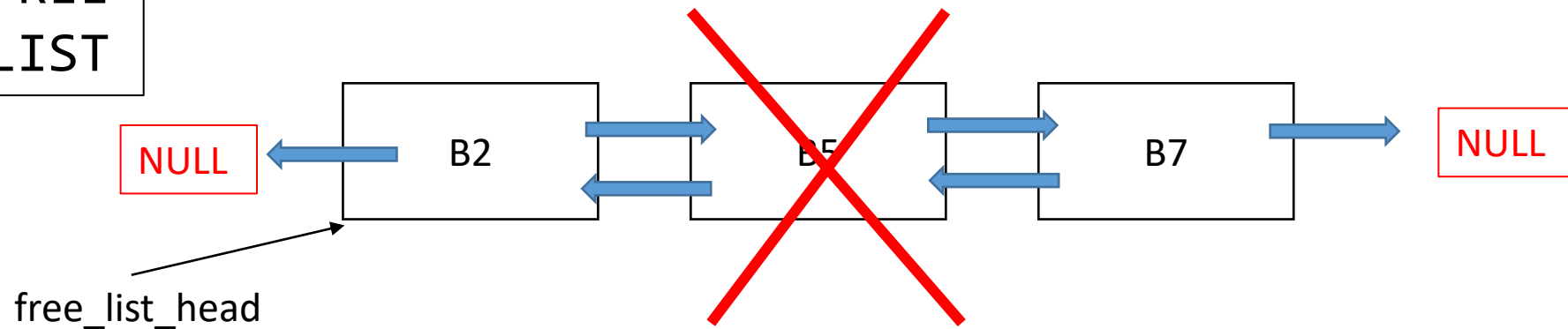


Free Block 4

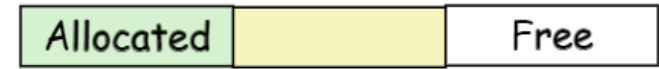


FREE LIST

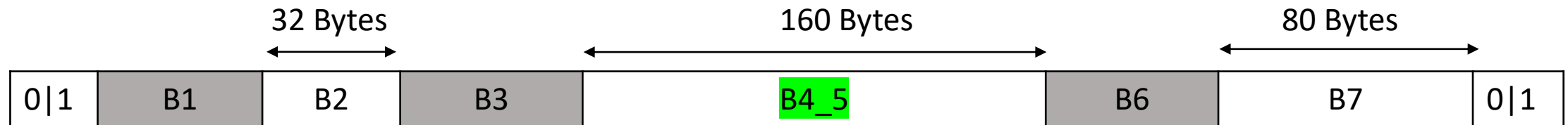
Step 1: Remove Block 5 from the Free List



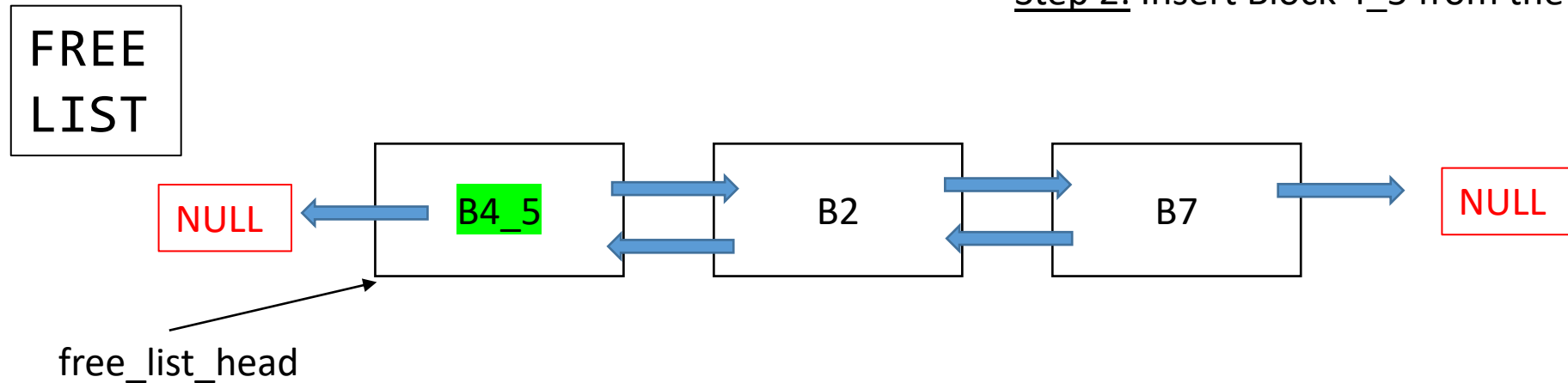
# Freeing A Block (Case 2)



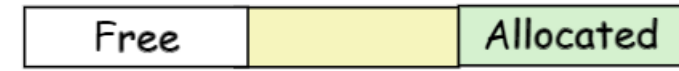
Free Block 4



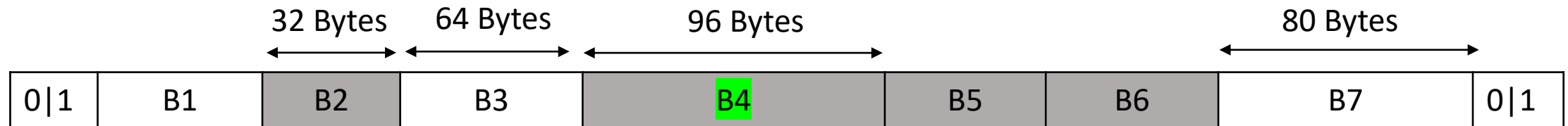
Step 2: Insert Block 4\_5 from the Free List



# Freeing A Block (Case 3)

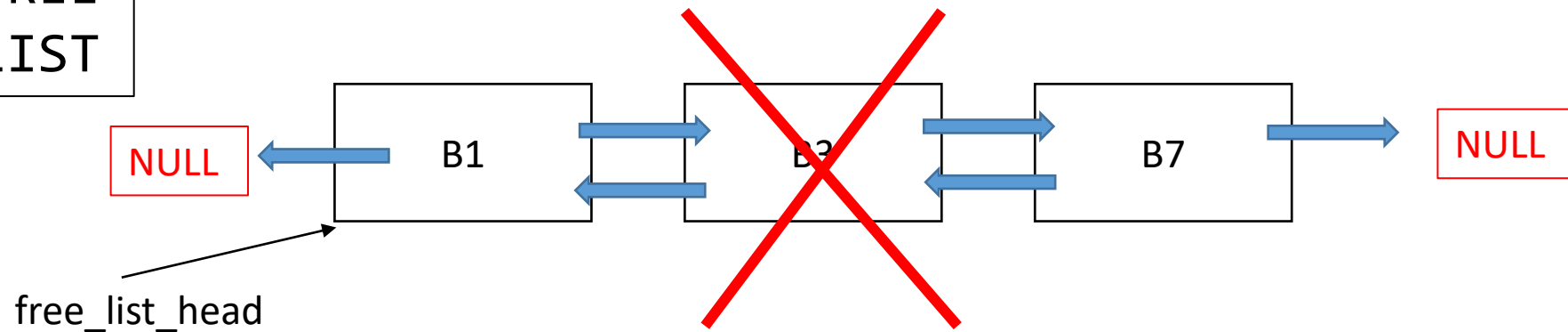


Free Block 4

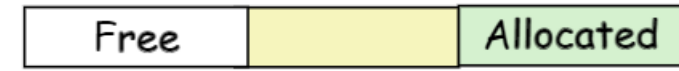


FREE  
LIST

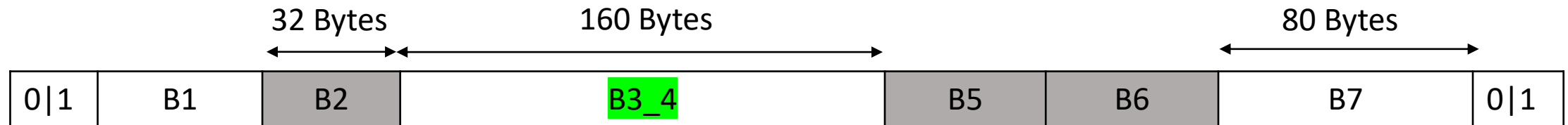
Step 1: Remove Block 3 from the Free List



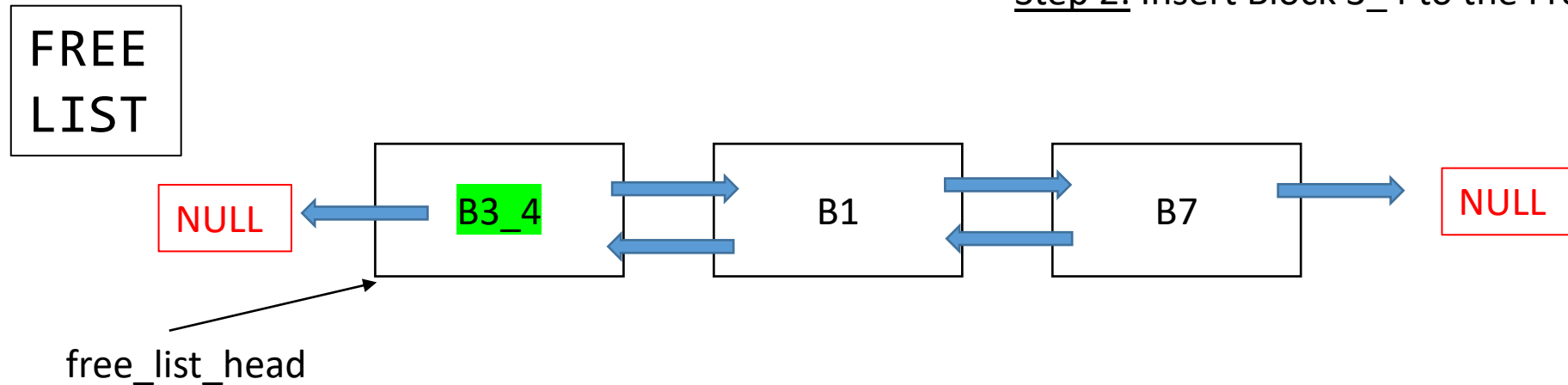
# Freeing A Block (Case 3)



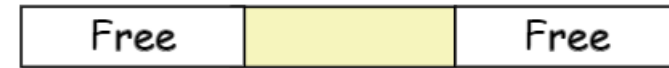
Free Block 4



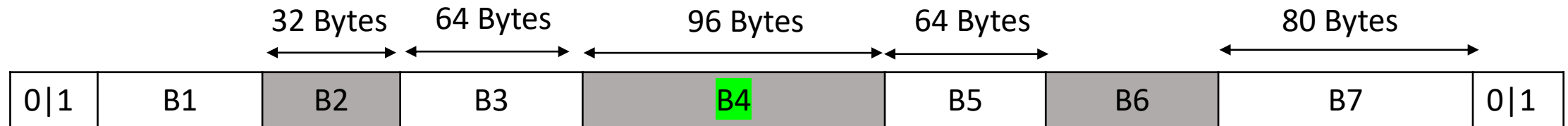
Step 2: Insert Block 3\_4 to the Free List



# Freeing A Block (Case 4)

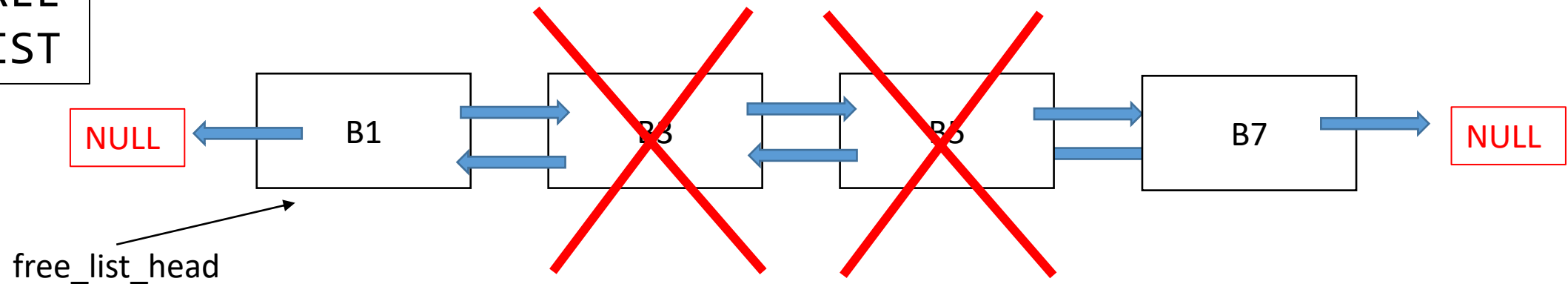


Free Block 4

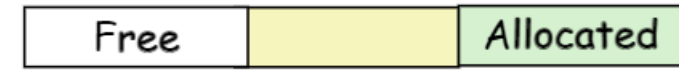


FREE LIST

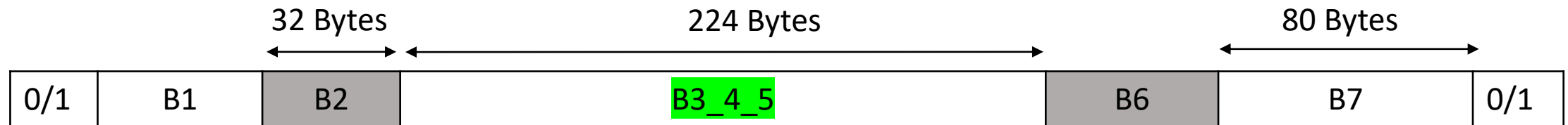
Step 1: Remove Block 3 and Block 5 from the Free List



# Freeing A Block (Case 4)

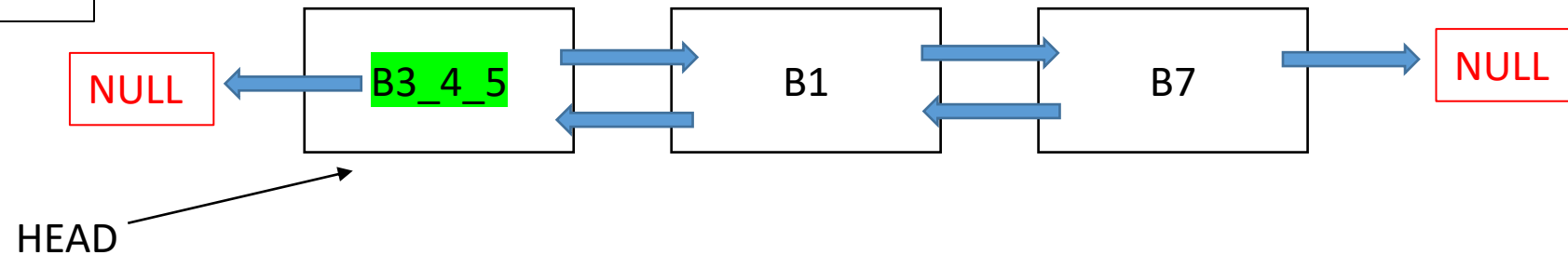


Free Block 4



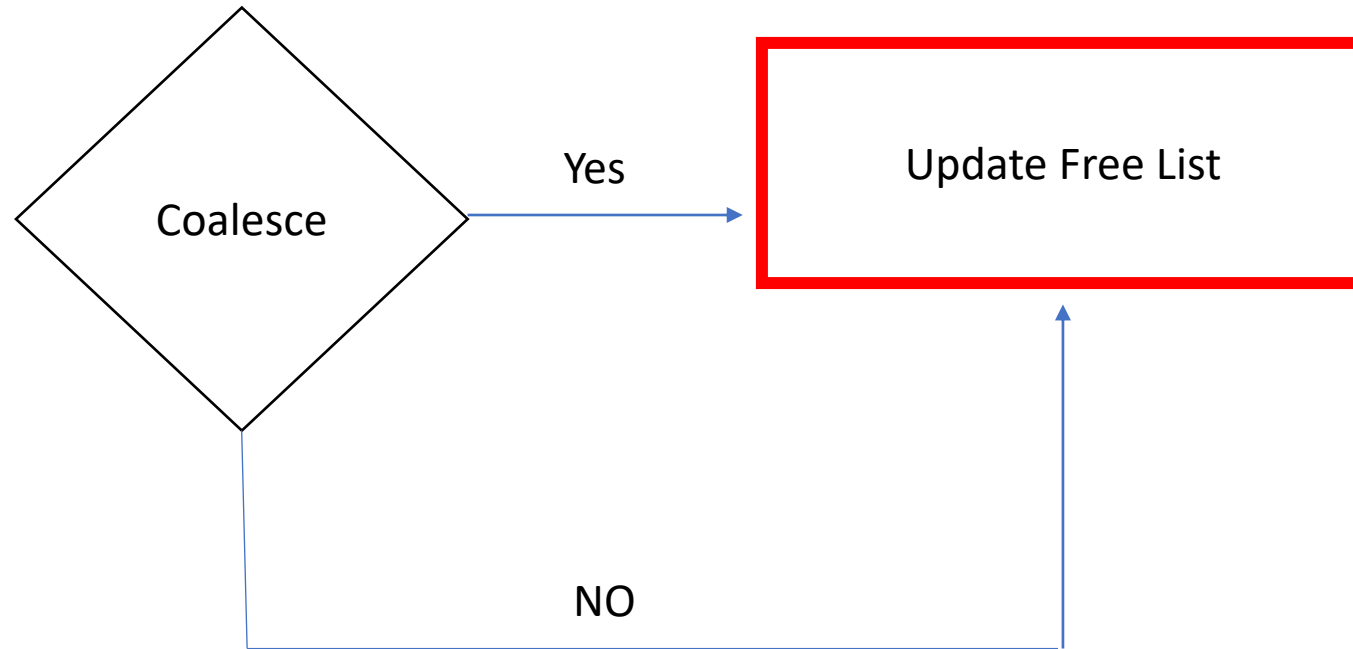
FREE  
LIST

Step 2: Insert Block 3\_4\_5 to the Free List





# mm\_free(void \*p)



# insert\_block(block\_t \*free\_block)

- Insert a Free Block to the Free List using LIFO
- Cases
  - What if the free list is empty?
  - What if the free list is Not empty?
  - Any other cases?

# remove\_block(block\_t \*free\_block)

- Cases
  - What if the Free list is empty?
    - Will you ever need to check this case in this function, ideally?
  - What if the block to be removed is the head?
  - What if the block to be removed is Not the head?
    - What if it's in the middle and what if it's in the end of the free list?
  - Any other cases?

# On Improving the Performance of Your Code

- Explicit Free List Implementation
- Modifying the Find\_Fit algorithm
  - Traverse only the Free Blocks
  - Find a hybrid of the different Find fit algorithms, if needed.
- Turn OFF (or comment out) all calls to `examine_heap()` and/or `check_heap()`

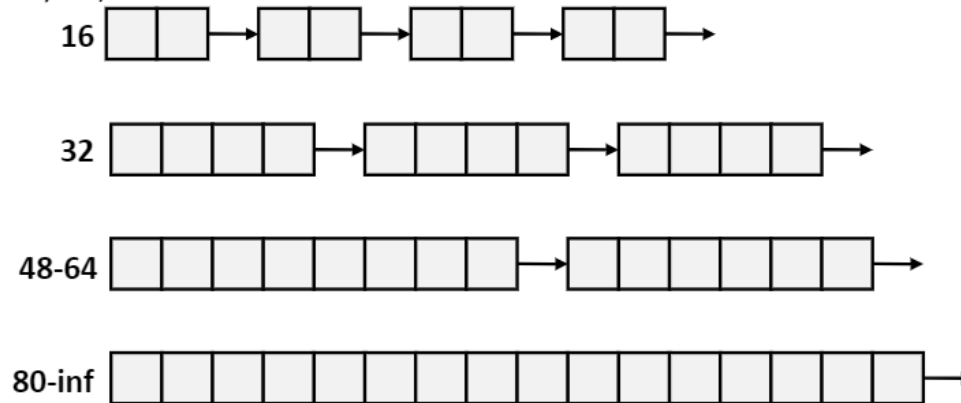
# On Improving the Performance of Your Code

## Segregated List (SegList) Allocators

---

- Each *size class* of blocks has its own free list
- Organized as an array of free lists

Size class  
(in bytes)



- Often have separate classes for each small size
- For larger sizes: One class for each two-power size

# Processes

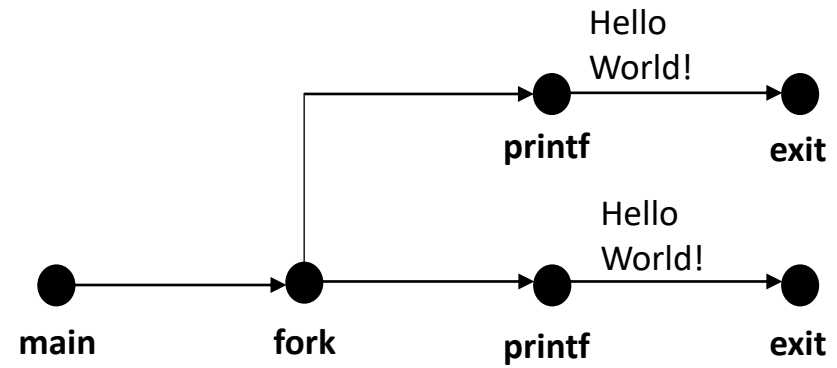
# fork()

- Creates a new process by **duplicating the calling process**.
- Return value
  - On **success**, the PID of the child process is returned in the **parent**, and 0 is returned in the **child**.
  - On **failure**, -1 is returned in the **parent**, no child process is created

# Example 1

```
int main()
{
    fork();

    printf("Hello world!\n");
    exit(0);
}
```





# Example 1

```
int main()
{
    fork();

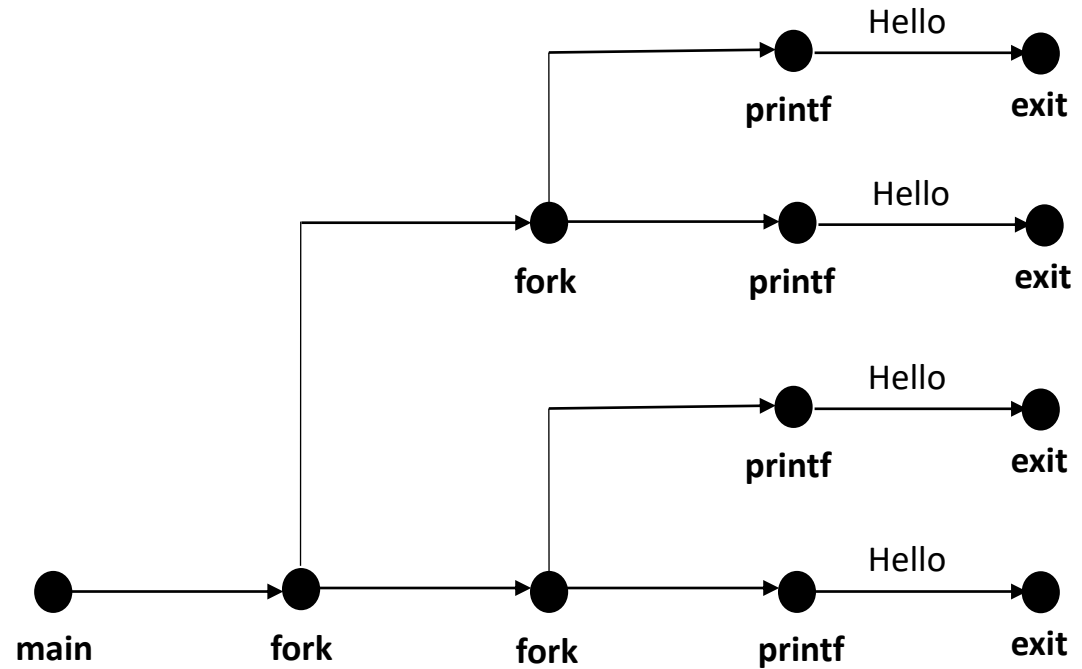
    printf("Hello world!\n");
    exit(0);
}
```

Output

Hello world!  
Hello world!

# Example 2

```
int main()
{
    fork();
    fork();
    printf("Hello\n");
    exit(0);
}
```



# Example 2

```
int main()
{
    fork();
    fork();
    printf("Hello\n");
    exit(0);
}
```

## Output

```
Hello
Hello
Hello
Hello
```

# wait()

- A call to wait() blocks the calling process until one of its child processes exits
- Return value:
  - On success, returns the process ID of the terminated child
  - On error, -1 is returned.

# Exercise

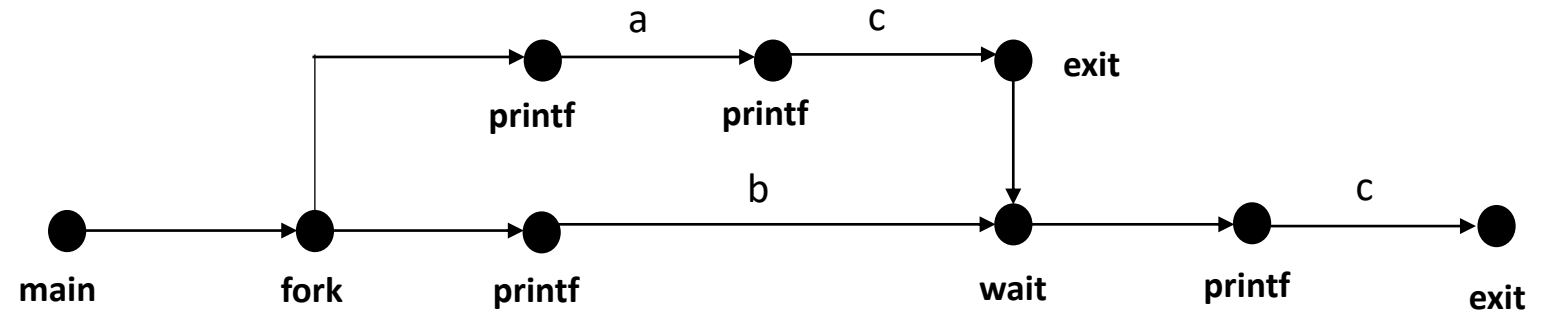
```
int main(){
    if (fork() == 0)
        printf("a");
    else{
        printf("b");
        wait(NULL);
    }
    printf("c");
    exit(0);
}
```

Can the following sequences be printed?

- acbc
- abcc
- bacc
- bcac
- cbca

# Exercise

```
int main(){
    if (fork() == 0)
        printf("a");
    else{
        printf("b");
        wait(NULL);
    }
    printf("c");
    exit(0);
}
```



# Exercise

```
int main(){
    if (fork() == 0)
        printf("a");
    else{
        printf("b");
        wait(NULL);
    }
    printf("c");
    exit(0);
}
```

Can the following sequences be printed?

- acbc - Yes
- abcc - Yes
- bacc - Yes
- bcac - No
- cbca - No

# Resource on wait

- A simple example of when wait() is needed
  - <https://www.youtube.com/watch?v=tcYo6hipaSA>



# exec()

- The exec() family of functions replaces the current process image with a **new process image**.
- The initial argument for these functions is the pathname of a file which is to be executed.

# Example

## ex1.c

```
int main(int argc, char *argv[])
{
    printf("PID of ex1.c = %d\n",
getpid());
    char *args = {"hello", NULL};
    execv("./ex2", args);
    printf("Next line of ex1.c");
    return 0;
}
```

## ex2.c

```
int main(int argc, char *argv[])
{
    printf("Inside ex2.c\n");
    printf("PID of ex2.c: %d\n",
getpid());
    return 0;
}
```

# Example

## Compilation

```
thoth$ gcc -o ex1 ex1.c  
thoth$ gcc -o ex2 ex2.c  
thoth$ ./ex1
```

## Output

```
PID of ex1.c = 5551  
Inside ex2.c  
PID of ex2.c: 5551
```