# Assignment 1

CS330: Operating Systems

*Total Marks: 100*

**Submission Deadline: 11.55PM, 5th September 2023**

## 1 Chain of Unary Operations [20 Marks]

In this question, you need to write three *C* programs `Part1/square.c`, `Part1/double.c` and `Part1/sqroot.c` performing square, double and square root operations, respectively on a non-negative integer. The generated executables (i.e., `square`, `double` and `sqroot`) of these programs can be chained in any pattern to perform composite operations on a given input. The order of the operations in the chained pattern would be from *left to right*.

### Syntax

```
$ ./<executable> <executable> .... <executable> <non-negative integer>
```

### Example

```
$ ./square sqroot double sqroot 8
4
```

Since, the order of the operations in the chained pattern is from **left** to **right**, the chained pattern should be viewed as:

```
sqroot(double(sqroot(square(8)))) = 4
```

### Output

Print the final result only (as shown in the example).

### Note

- At-least 1 unary operation and at-most 16 unary operations will be specified during testing.
- You can assume that the result of operations will always fit in a 8-byte integer type (i.e., unsigned long on 64-bit machines).
- If the square root of a number is a fraction then round off the result to the nearest integer. For example, 2.5 should be rounded to 3 and 2.4 should be rounded to 2.

### Error handling

In case of any error, print **"Unable to execute"** as output.

## System calls and library functions allowed

You **must only use** the below mentioned APIs to implement this question.

```
- fork                  - malloc
- exec* family          - free
- strcpy                - strcat
- strcmp                - strto* family
- ato* family           - wait/waitpid
- printf, sprintf       - sqrt
- round                 - exit
- strlen
```

## Testing

Script `Part1/run_tests.sh` contains 3 sample test cases. Run this script to check whether your implementation passes the test cases or not. A sample output after running the script would be:
Test 1 is Passed
Test 2 is Passed
Test 3 is Passed

## 2 Directory Space Usage [35 Marks]

In this question, you have to write a program (`myDU.c`) that finds the space used by a directory (including its files, sub-directories, files in sub-directories, sub-sub directories etc.). Let's call this directory as the *Root* directory.

### Syntax

```
$./myDU <relative path to a directory>
```

### Example

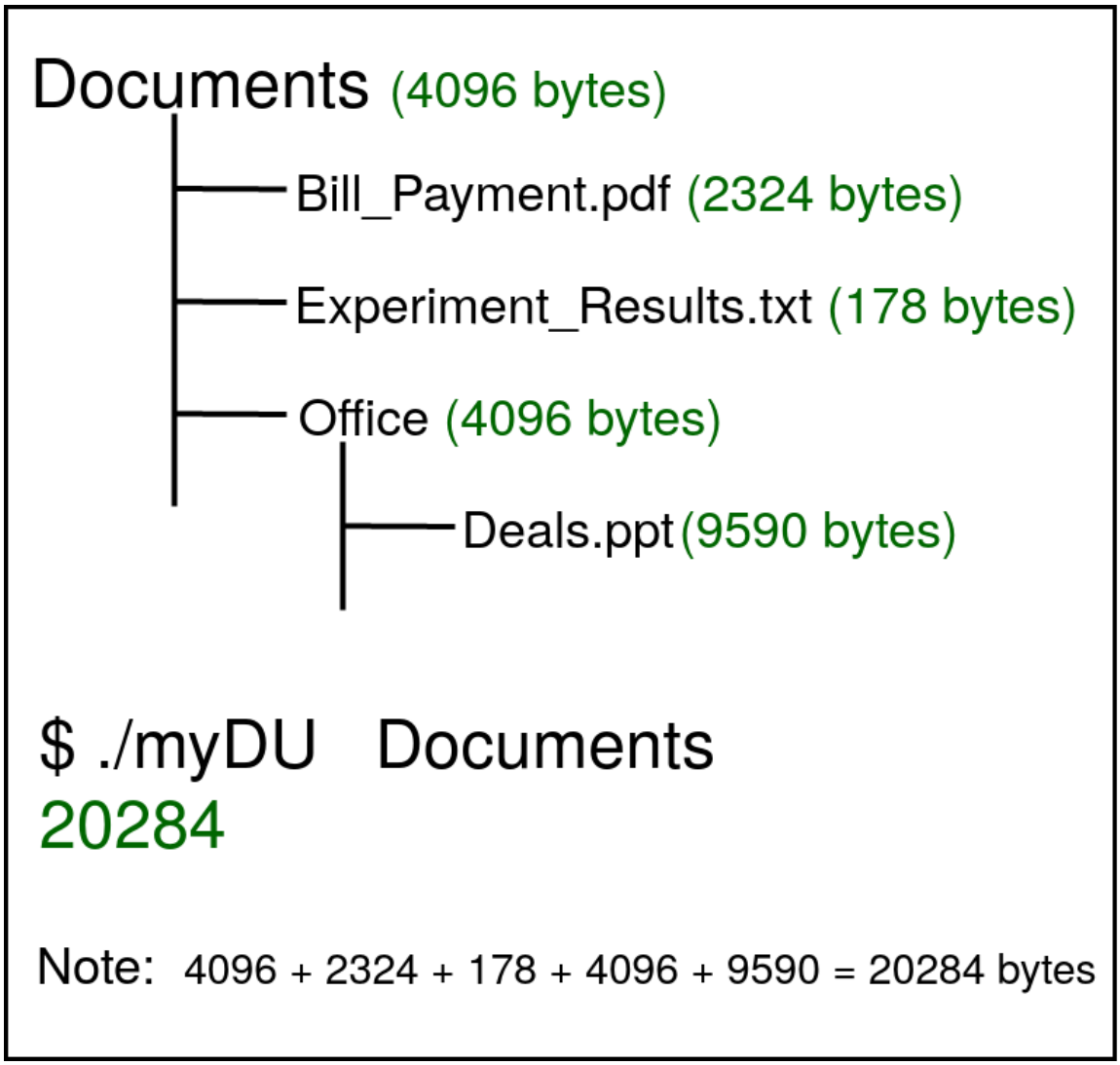


Figure 1: Example to illustrate the use of directory space usage finding utility

Figure 1 shows the structure of a directory called **Documents** which is the designated *Root* directory in this example. This directory contains files such as **Bill_Payment.pdf**, **Experiment_Results.txt** and a sub-directory called **Office**. The Sub-directory **Office** contains a file named **Deals.ppt**. To find the size of the **Documents** directory, its name is passed to your directory space usage finding utility as **($./myDU Documents)**. Your utility is expected to print the total size of the contents of the passed root directory in bytes **(For eg: 20284)**. Note that, this is inclusive of directory sizes.

## Output

Only print the size of the *Root* directory in bytes (Refer to Figure 1)

## Detailed instructions

To make the calculation process more efficient, we propose a method where different sub-directories under the *Root* directory will be processed by different processes. The exact working is detailed in the following points.

- Assume that there are N immediate sub-directories under the *Root* directory. For each immediate child sub-directory under the provided *Root* directory, your program must create a new process $P_i$ ($i$ will range from 1 to the total number of immediate sub-directories under *Root*). Each child process $P_i$ should find the size of the $i^{th}$ child sub-directory (including all files/directories under it and the size of the sub-directory itself) and pass this information back to the parent process. Parent process should find the size of the files immediately under it along with the *Root* directory size. Finally, parent process will find the sum of all sizes and print the final output.

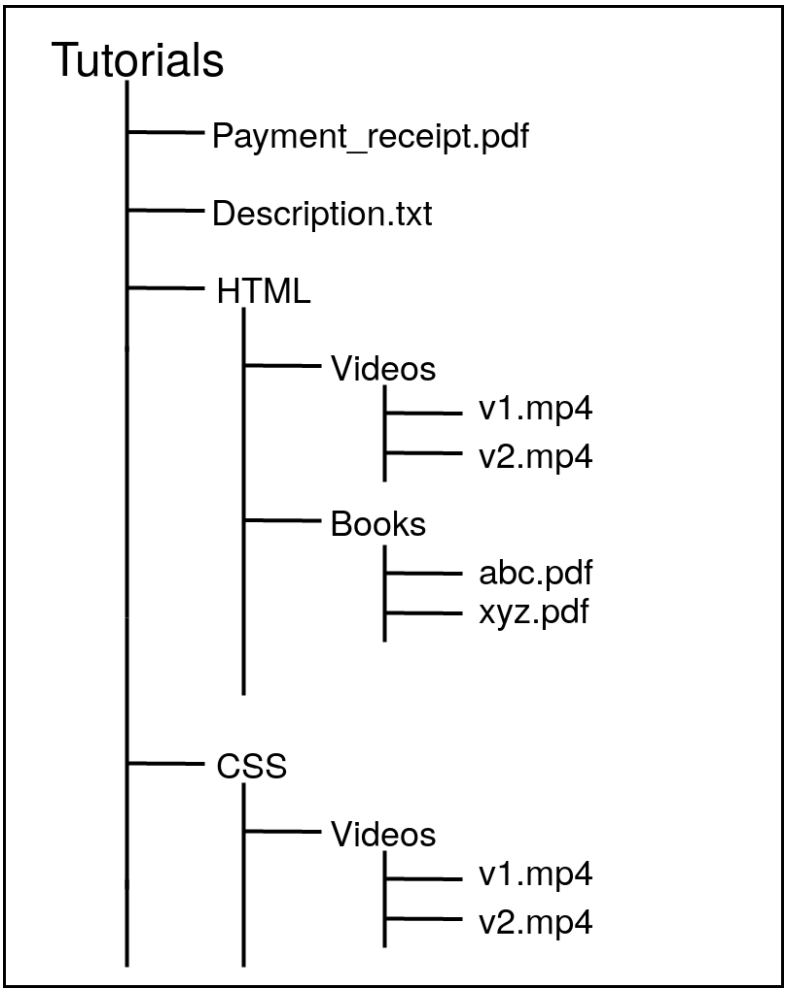


Figure 2: Sample directory structure

For example: In Figure 2, there are two immediate sub-directories (**HTML, CSS**) under the *Root* directory (**Tutorials**). In this case, the parent process should create two child processes, say, $P_1$ and $P_2$. $P_1$ should calculate the size of the sub-directory (**HTML**) which is a sum of sizes of all files and directories under **HTML** including the size of **HTML** itself. Similarly, $P_2$ should calculate size of the directory **CSS**. Both $P_1$ and $P_2$ should return the result back to the parent using pipes. Finally, the parent process would find the size of the files immediately under

it (**Payment_receipt.pdf, Description.txt**) and the size of the **Tutorials** directory to report the final result.

- Your program should *only* use pipes (pipe() system call) to communicate between parent process and the child processes. There is no restriction on the number of pipes to be used.

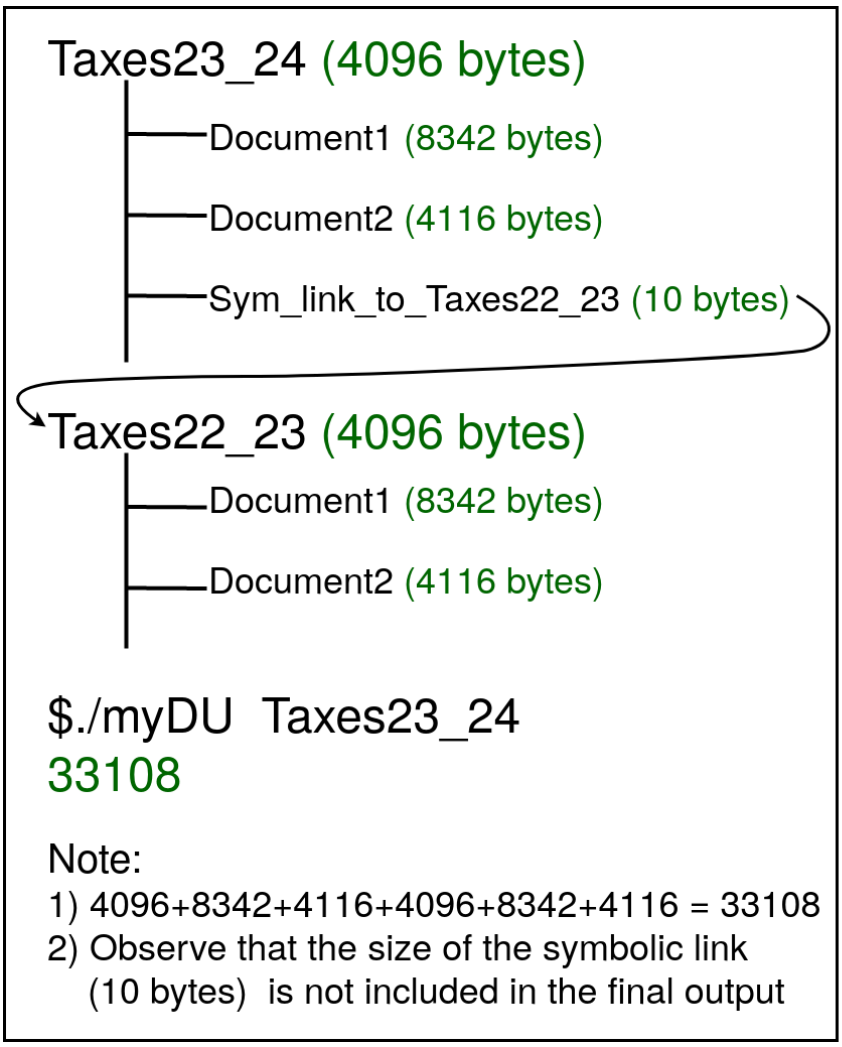


Figure 3: Example to illustrate the handling of symbolic links by ./myDU utility

- A symbolic link is a special type of file in Linux that points to another file or directory. Symbolic links can be present anywhere in the *Root* directory tree. You should resolve the symbolic links and find the size of the file/directory pointed by a symbolic link instead of reporting the size of the symbolic link file (Refer figure 3). It can be assumed that a symbolic link will never point to itself recursively. For example, in Figure 3, `Taxes22_23` directory will not contain a symbolic link that points to the `Taxes23_24` directory or directly to the `Sym_link_to_Taxes22_23` symbolic file in the `Taxes23_24` directory.

- It is the responsibility of the parent process to find the size of the file/directory pointed by a symbolic link which is present immediately under *Root* directory. For example, in Figure 3 parent process will not create any child process. However, symbolic links present in a sub-directory should be processed by the child process handling the sub-directory.

- During testing, only relative path of the *Root* directory will be passed to the `./myDU` utility. It can be assumed that the size of the directory path generated during testing will not exceed 4096 bytes.

- You can assume that the total size of the *Root* directory will fit in a 8-byte integer type (i.e., *unsigned long* on 64-bit machines)

## Error handling

In case of any error, print **"Unable to execute"** as output.

## System calls and library functions allowed

```
- fork                  - malloc
- exec* family          - free
- pipe                  - stat
- opendir               - lstat
- readdir               - readlink
- closedir              - strlen
- read                  - open
- write                 - close
- strcpy                - strcat
- strcmp                - strto* family
- ato* family           - wait/waitpid
- printf family         - exit
- dup                   - dup2
```

## Testing

3 sample test cases have been provided in the `Part2` directory. Run the `test.sh` script (`$./test.sh`) to compare your `./myDU` utility's output with the expected output of the sample testcases.

# 3 Dynamic memory management library[45 Marks]

In this question you have to implement two library functions **memalloc()** and **memfree()** that are equivalent to the *C* standard library functions `malloc` and `free`. The **memalloc()** call will be used for dynamic memory allocation and the **memfree()** call will be used to free the memory previously allocated by **memalloc()**.

## Syntax

```
void *memalloc(unsigned long size);
int memfree(void *ptr);
```

## Detailed instructions

- `memalloc()` takes single argument of type `unsigned long`. This argument specifies the amount of memory requested from `memalloc()` in bytes. The memory allocated by `memalloc()` should not be initialized. `memalloc()` returns a generic pointer (`void*`) to the allocated memory. If 0 is passed as the argument to the `memalloc()` or `memalloc()` cannot satisfy the request, return `NULL`.
- `memfree()` takes a single argument of type `void*`. This argument is a pointer to the memory allocated by `memalloc()` that is to be freed. `memfree()` returns 0 on success. You can assume that a valid address (which has been returned by `memalloc()` and hasn't been freed before) will be passed to `memfree()`.
- If `memalloc()` has to request memory from the OS, it must be done using `mmap()` with size being multiple of 4MB. `memalloc()` serves memory allocation requests of the user-space programs from these allocated chunks of memory. If a request to `memalloc()` is larger than 4 MB, request a smallest multiple of 4 MB memory chunk from OS that satisfies the requested memory size. You can assume that `mmap()` will never fail due to lack of memory.

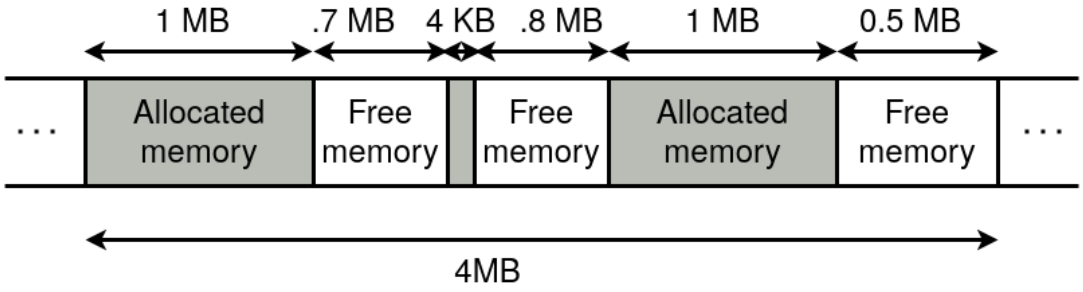


Figure 4: Possible state of the memory managed by the `memalloc()/memfree()`

- Figure 4 illustrates the possible state of the memory managed by the `memalloc()/memfree()`

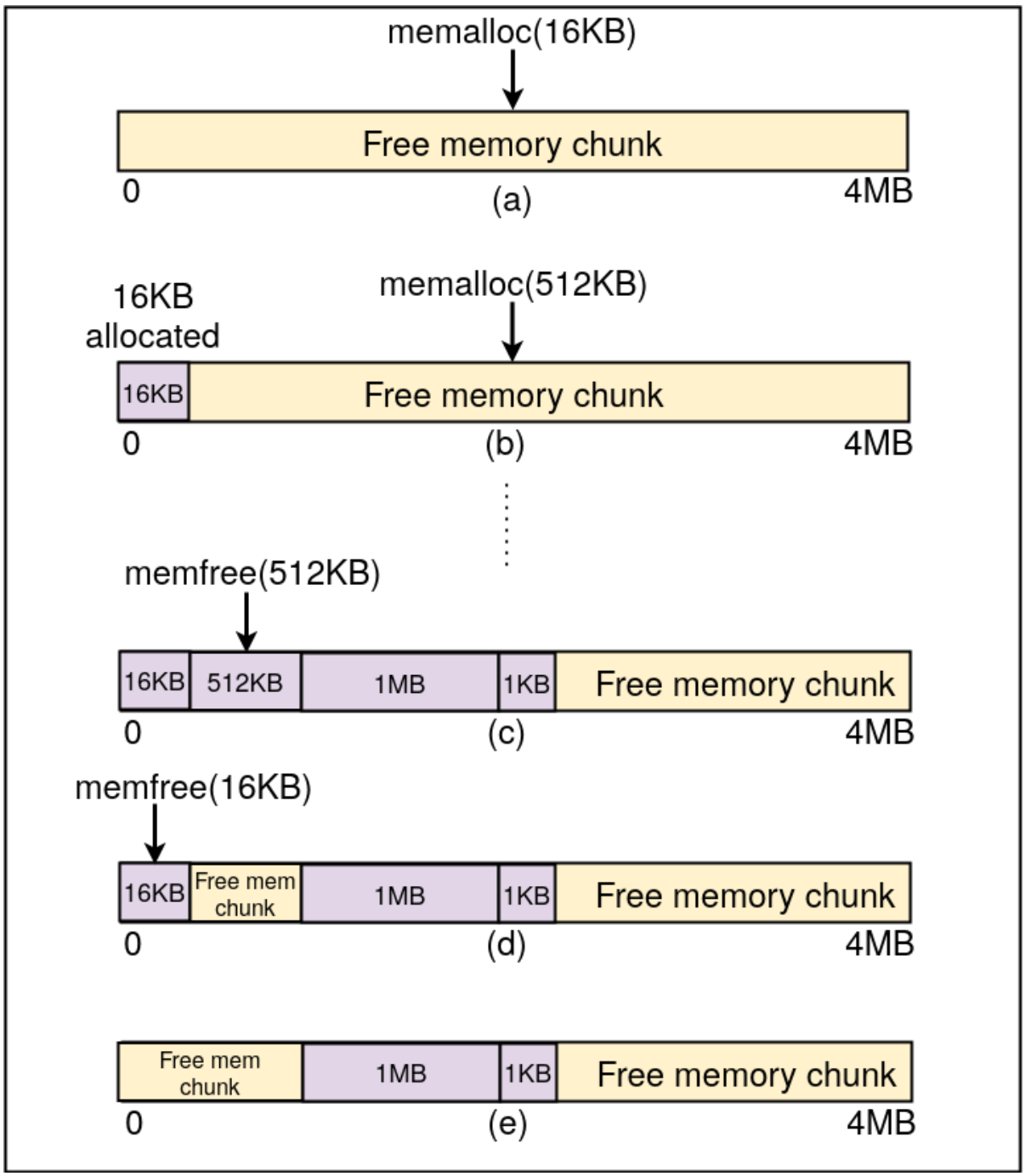


Figure 5: Example to illustrate the allocation and freeing of memory using memalloc() and memfree()

- Figure 5 illustrates how requests for allocation and freeing of memory can be served by `memalloc()` and `memfree()` from the memory (4MB chunk) obtained from OS.
- To handle the allocation and de-allocation requests you need to maintain some metadata corresponding to the allocated and free memory chunks. For example, metadata can help to identify if a free memory chunk is large enough to serve a memory allocation request made by the user-space program.

## Metadata

- For correct working of the library routines, you have to maintain metadata to keep track of the allocated and free memory regions within the larger chunk of memory obtained from the OS. Metadata is maintained **within** the allocated and free memory chunks itself (see Figure 6).

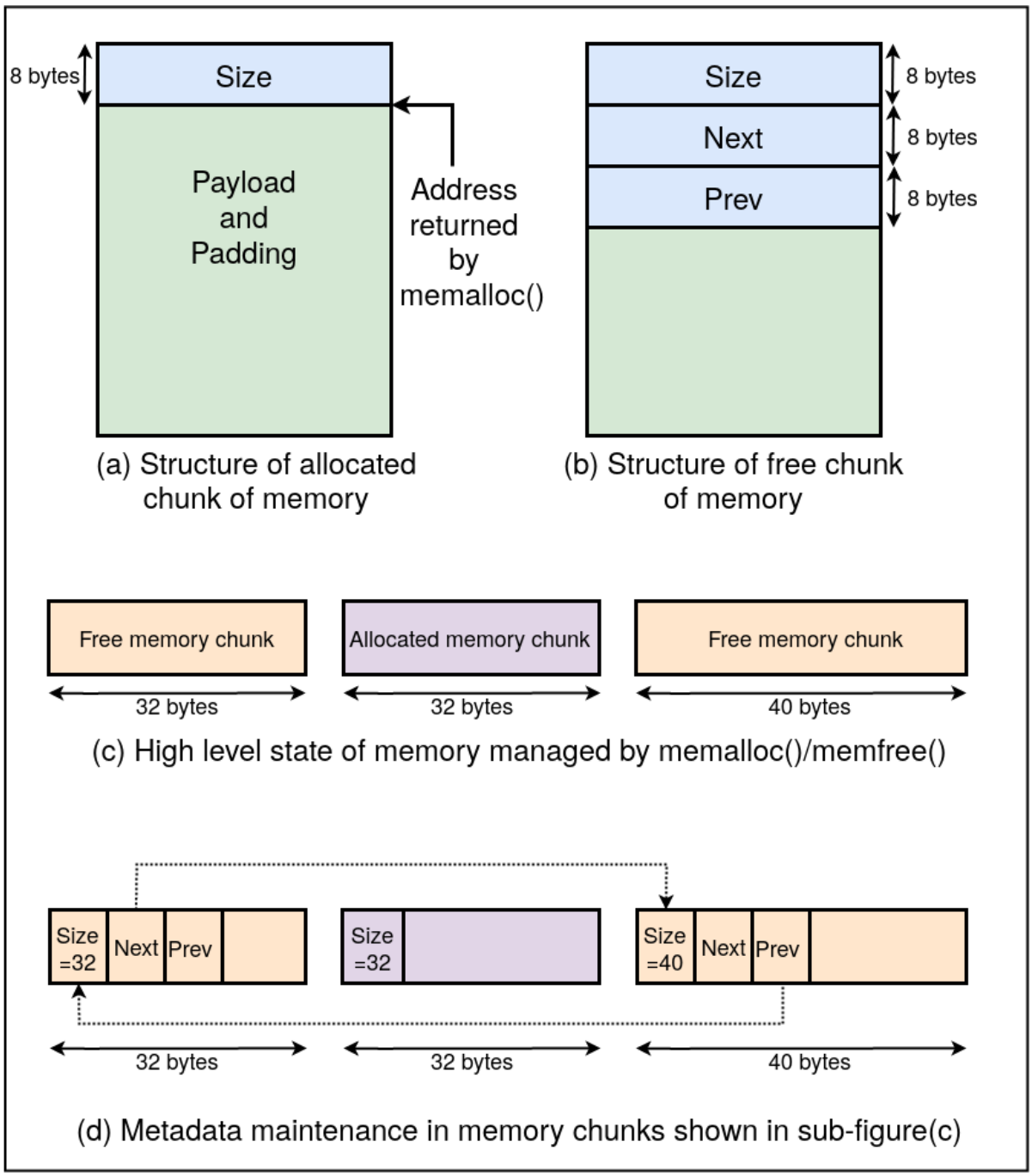


Figure 6: Example to illustrate the metadata maintenance corresponding the allocated and free memory chunks

- As shown in 6`(a),(c),(d)`, `Size` contains the size (in bytes) of the memory chunk allocated by `memalloc()` on the request of the userspace application. `Size` is stored in the first 8 bytes of the allocated memory chunk. Allocated memory chunk should be large enough to store metadata (`Size`) as well as to fulfill the memory size requested by userspace program from the `memalloc`.

- Free memory is maintained in the form of a doubly linked list where the head of the list is a global variable. Let us refer to this list as the `free list`. As shown in 6`(b),(c),(d)`, `Size, Next, Prev` are stored within each free chunk of memory present in the `free list`. `Size` contains the size (in bytes) of the free memory chunk. `Next and Prev` are the pointers to the next free chunk of memory and the previous free chunk of memory in the `free list`. Each of these pointers is stored in 8 bytes.

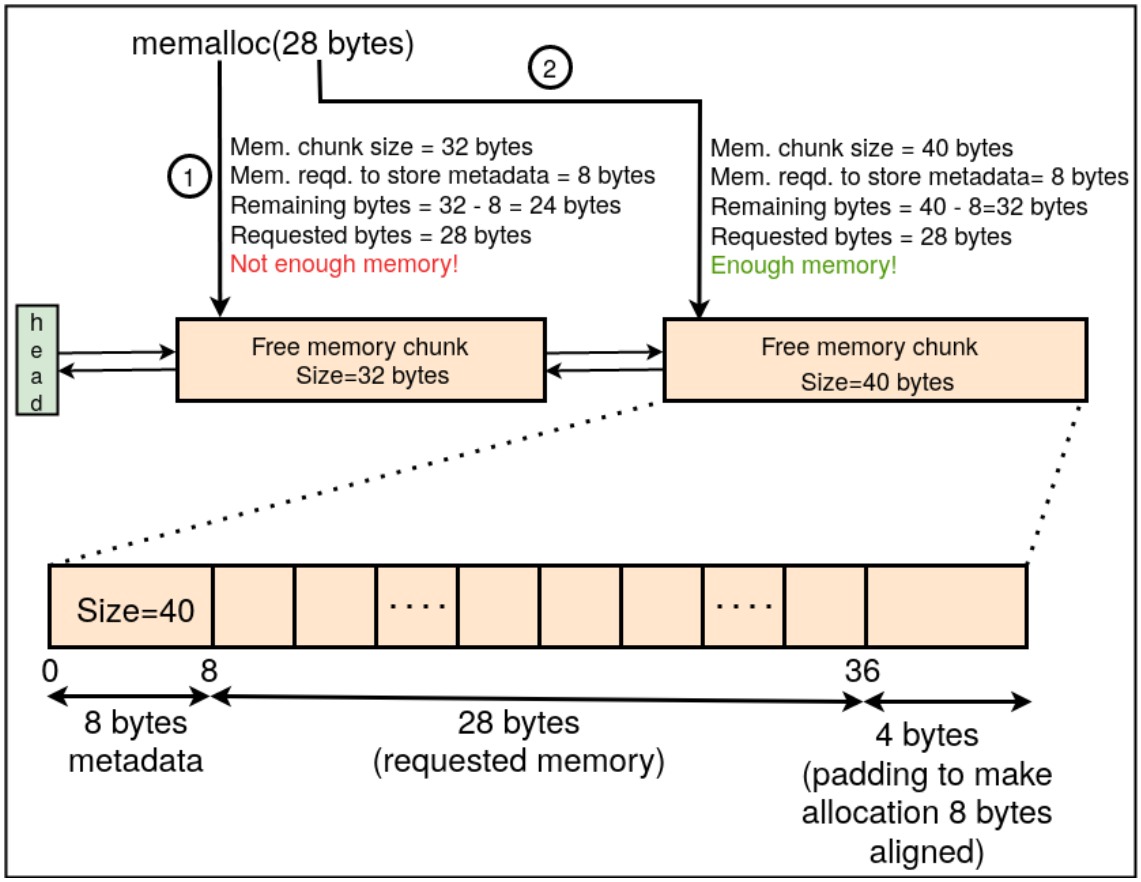


Figure 7: Example to illustrate memory allocation using first fit approach and the usage of padding

## Memory allocation

- When a request for memory allocation comes from the userspace program to `memalloc()`, the request should be served using the *First Fit* approach. The first fit approach specifies that when a request of size $S$ is made, `memalloc` should find the first free chunk of memory in the `free list` which is large enough to service the request.

  For example, in Figure 7, a request of size 28 bytes is made by the user-space program. The `memalloc()` logic checks the size of the first free memory chunk (32 bytes) and concludes that this memory chunk is not large enough to service the request. So, it checks the next free memory chunk in the free list (40 bytes chunk) and concludes that the request can be served using this memory chunk.

- Chunk of memory chosen to serve the request should be a multiple of 8 bytes. Let us call the extra bytes allocated to serve the request as a multiple of 8 bytes as *eight-bytes alignment padding*. For example, in Figure 7, a padding of 4 bytes is used make the memory allocation 8 bytes aligned.

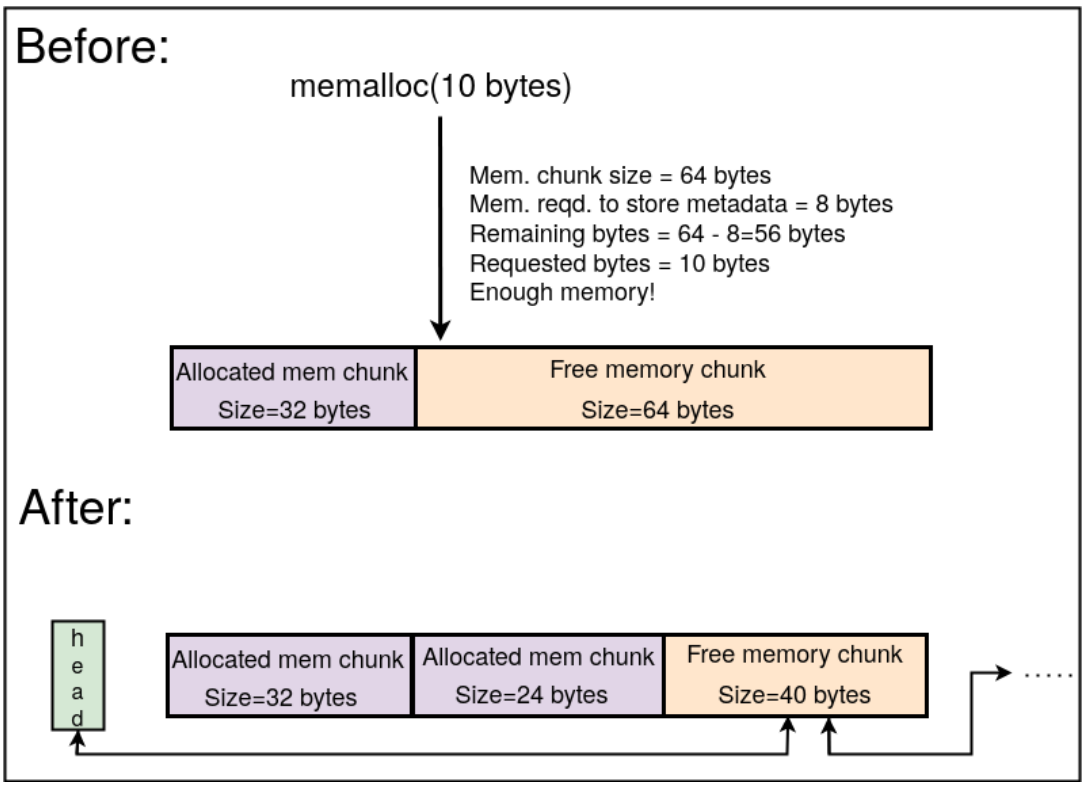


Figure 8: Example to illustrate the splitting of a free memory chunk into smaller chunks

- If the free memory chunk is exactly the same as the request size + metadata size + 8 bytes alignment padding, the free memory chunk is simply removed from the free list. For example: In Figure 7, entire free memory chunk of size 40 bytes will be removed from the free list.
- If the free memory chunk is larger than the request size + metadata size + 8 bytes alignment padding by `'b'` bytes, then the remaining `'b'` bytes should be handled as follows,
  `Case 1:` If `'b'` $< 24$, include these `'b'` bytes in the memory allocated to the user-space application as extra padding. Thus, full free memory chunk is allocated just as if it had been exactly the right size.
  `Case 2:` If `'b'` $\geq 24$, the free memory chunk is split into two memory chunks. The chunk on the left (which should be of the size request size + metadata size + 8 bytes alignment padding) should be allocated to the user-space application and removed from the free list. The chunk on the right should be added to the free list with appropriate adjustment to size and other meta-data. The chunk of memory added back to the free list should be inserted at the head of the free list. For example, In Figure 8, chunk of size 64 bytes is split into two smaller chunks to serve a user invocation of **memalloc(10)**.

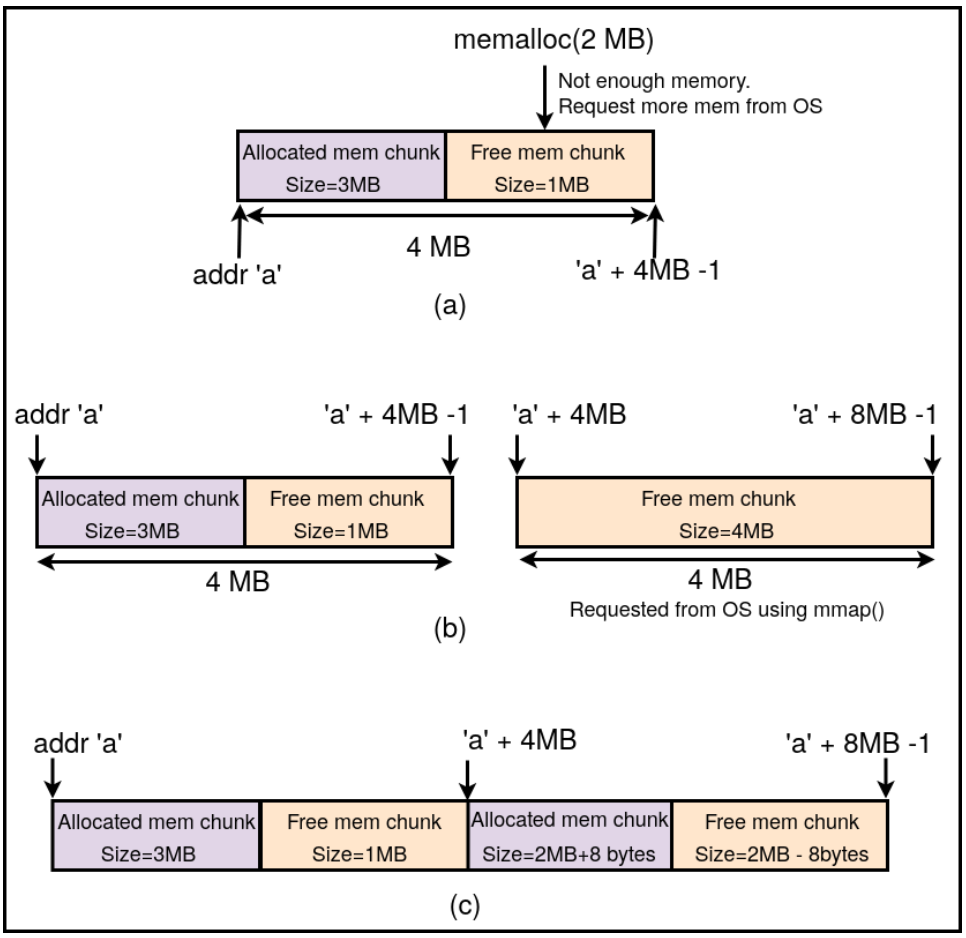


Figure 9: Example to illustrate the case when extra memory is requested from OS

- When no available free memory chunk can satisfy the applications memory request, `memalloc` should issue a `mmap` system call to the OS to allocate a new free memory chunk of size equal to the smallest multiple of 4 MB that is large enough to service the request. For example, In Figure 9`(a)`, a request of 2MB memory size can't be served by the `memalloc()`. So, it requests 4MB memory from the OS (Figure 9`(b)`).
- If a free memory chunk (already present in free list) and a newly allocated region using `mmap` system call are contiguous, then no coalescing (combining/merging) of the two memory regions should be done before serving the request made by the application. For example, in Figure 9`(b)`, memory chunk at address `'a'+ 3MB` to `'a' + 4MB - 1` and the newly allocated memory chunk at address `'a'+ 4MB` to `'a' + 8MB - 1` are contiguous. The 2MB memory allocation request should be served from newly allocated memory chunk (address `'a'+ 4MB`), as shown in Figure 9`(c)`, instead of coalescing both free chunks of memory into a single free memory chunk and serving request from the address `'a'+ 3MB`.

## Memory Deallocation

When freeing a memory chunk using `memfree` (the chunk is refereed to as `'f'`), following cases may arise:

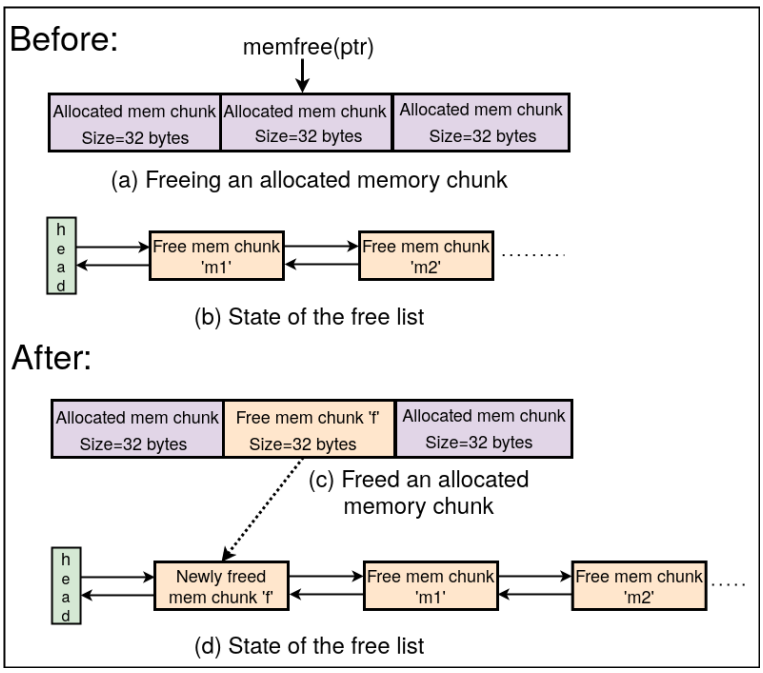


Figure 10: Example to illustrate the freeing of a memory chunk such that the memory chunks on its left and right side are not free (Case 1)

`Case 1:` Contiguous memory chunks on the left side and the right side of `'f'` are allocated. In this case, simply insert `'f'` into the head of the free list. Figure 10 demonstrates the working logic.

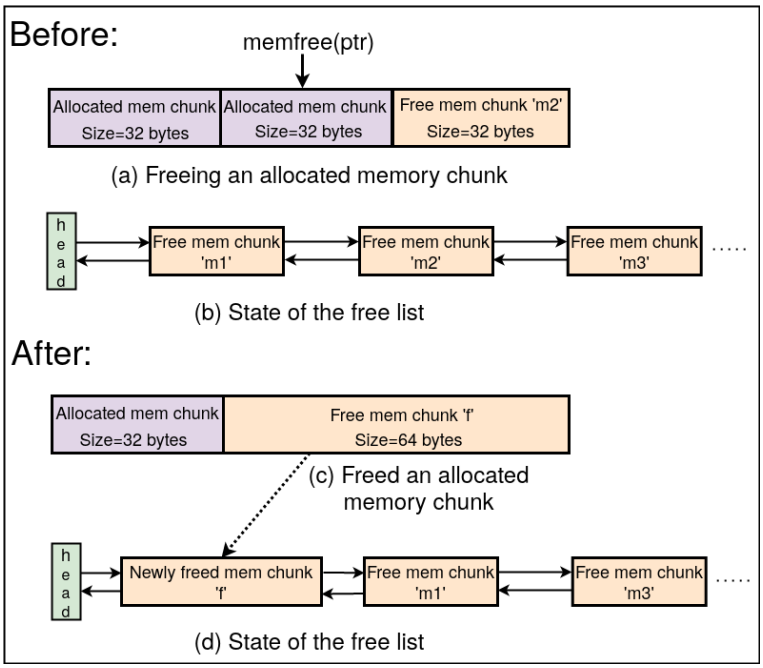


Figure 11: Example to illustrate the freeing of a memory chunk such that the memory chunk on its right is free (Case 2)

`Case 2:` Contiguous memory chunk on the right side of `'f'` is free. In this case, coalesce (combine/merge) `'f'` and memory chunk on the right side together into a single free memory chunk. The newly coalesced memory chunk should be inserted to the head of the free list. Figure 11 demonstrates the working logic.

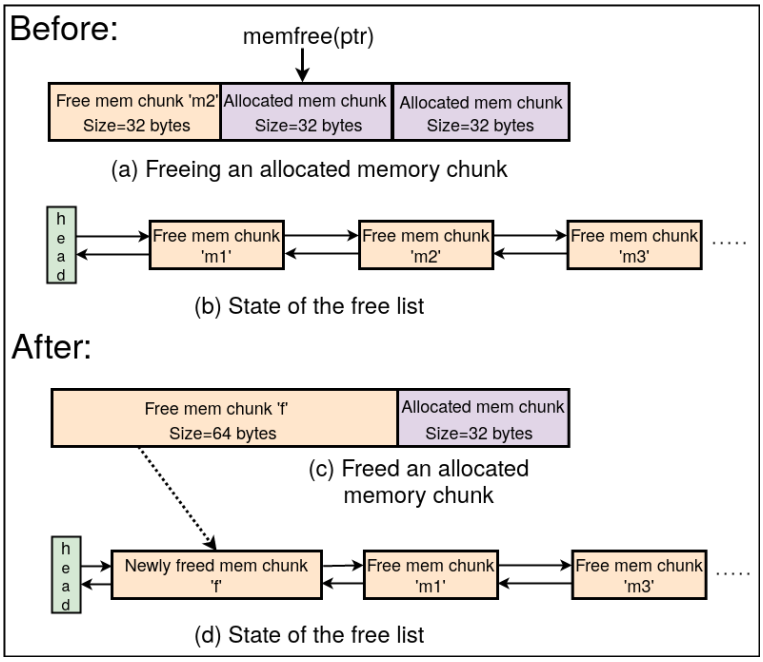


Figure 12: Example to illustrate the freeing of a memory chunk such that the memory chunk on its left is free (Case 3)

`Case 3:` Contiguous memory chunk on the left side of `'f'` is free. In this case, coalesce `'f'` and memory chunk on the left side together into a single free memory chunk. The newly coalesced memory chunk should be inserted to the head of the free list. Figure 12 demonstrates the working logic.

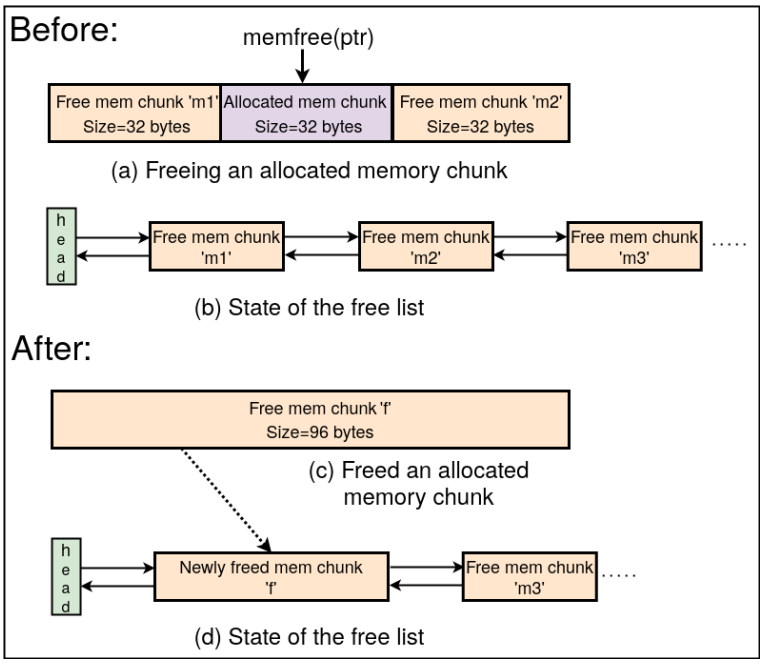


Figure 13: Example to illustrate the freeing of a memory chunk such that the memory chunks on its left and right side are free (Case 4)

`Case 4:` Contiguous memory chunks on the both left side and the right side of `'f'` are free. In this case, coalesce `'f'` with both the memory chunks on the left side and the right side together into a single free memory chunk. The newly coalesced memory chunk should be inserted to the

head of the free list. Figure 13 demonstrates the working logic.

*Note:* Memory requested by `memalloc()` from OS using `mmap` system calls will not be returned back to the OS when `memfree()` is called.

## Error handling

In case of any error during `memalloc()`, return `NULL`. In case of any error during `memfree()`, return `-1`.

## System calls and library functions allowed

- `mmap`

*Note:* You are *not* allowed to use any memory allocation/de-allocation helpers in the standard library such as malloc(), free(), calloc(), realloc() in your implementation.

## Testing

- In this question, 3 files are involved
  - Library file containing definitions of `memalloc()` and `memfree()`. This file is present as `Part3/mylib.c`. You are expected to write your code for `memalloc()` and `memfree()` implementation in this file.
  - Header file containing declarations of `memalloc()` and `memfree()`. This file is present as `Part3/mylib.h`. You don't need to modify this.
  - Test program file containing `main()` function. Test programs are present in `Part3/Testcases`. You can create your own test cases in this directory.
- 3 sample test cases have been provided in the `Part3/Testcases` directory.
- Makefile (`Part3/Makefile`) has been provided to automatically perform the steps of compiling and linking test cases with your library to generate the test cases execeutables. Change your current working directory to `Part3` directory and run the Makefile with the following command: `$make` `Note:` After making code changes in your library code (`Part3/mylib.c`) or test cases, you have to run Makefile again before running the test case executable.
- Change your current working directory to `Part3/Testcases` and run the test case executable (`for example, $./test1`) to check whether your code passes the test case or not.
- You can create your custom test cases in `Part3/Testcases` directory and run Makefile to generate the test cases executables.

# 4 Submission

- Make sure that your implementation doesn't print unnecessary data. Your output should match exactly with the expected output specified in each question.
- You have to include a file named '`declaration`' in your submission. In the '`declaration`' file, you have add the following statement:

  "`I have read the CSE department's anti-cheating policy available at https://www.cse.iitk.ac.in/pages/AntiCheatingPolicy.html. I understand that plagiarism is a severe offense. I have solved this assignment myself without indulging in any plagiarism. If my submission is found to be plagiarized from the internet, fellow students, etc., then strict action can be taken against me. <Your Name and Roll No>`"
- In the '`declaration`' file, you also have to mention the resources, such as websites, open source content you referred to while solving this assignment.
- You have to submit zip file named `your_roll_number.zip` (for example: `1211405.zip`) containing **only** the following files in specified folder format:
  - `YourRollno/Part1/square.c`, `YourRollno/Part1/double.c`, `YourRollno/Part1/sqroot.c`
  - `YourRollno/Part2/myDU.c`
  - `YourRollno/Part3/mylib.c`
  - `YourRollno/declaration`
- If your submission is not as per the above instructions, a penalty of 20 marks will be applied on the marks obtained in this assignment.
- **Note:** No code changes will be allowed after the assignment submission period has ended. So, test your implementation thoroughly with the provided test cases as well as your custom test cases.