

Lab-08: Memory Management

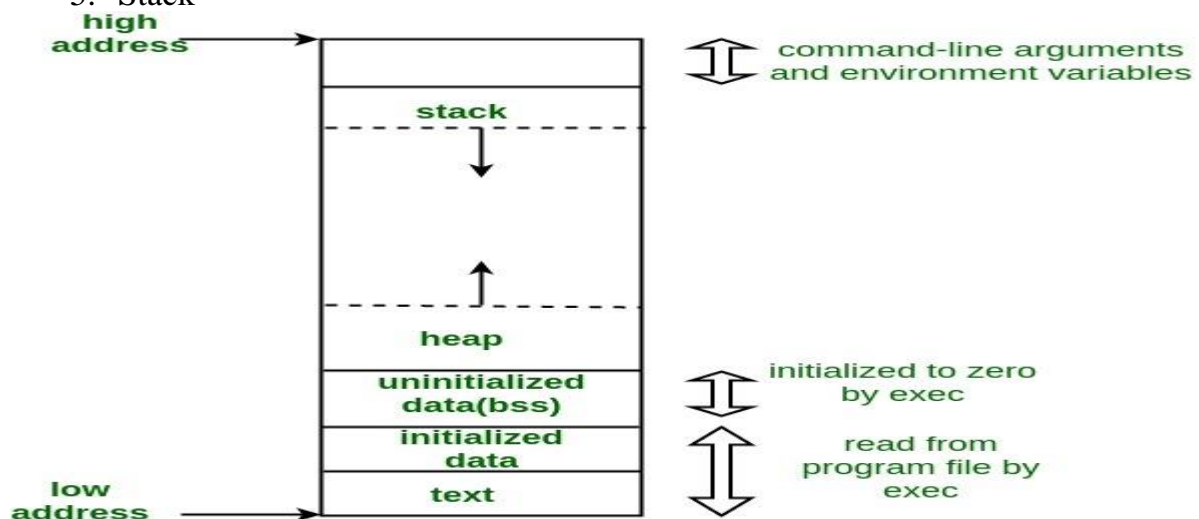
1. Objectives

- Understand the typical use cases and advantages of `mmap()`.
- Gain experience with different `mmap()` uses.

2. Memory Layout of C Programs

A typical memory representation of a C program consists of the following sections.

1. Text segment (i.e. instructions)
2. Initialized data segment
3. Uninitialized data segment (bss)
4. Heap
5. Stack



A typical memory layout of a running process

- **1. Text Segment:** A text segment, also known as a code segment or simply as text, is one of the sections of a program's memory which contains executable instructions. As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions. If you're interested in memory management and optimization techniques, the [C Programming Course Online with Data Structures](#) offers a complete guide to memory handling in C.
- **2. Initialized Data Segment:** Initialized data segment, usually called simply the Data Segment. A data segment is a portion of the virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.
- Note that, the data segment is not read-only, since the values of the variables can be altered at run time.
- This segment can be further classified into the initialized read-only area and the initialized read-write area.

- For instance, the global string defined by `char s[] = "hello world"` in C and a C statement like `int debug=1` outside the main (i.e. global) would be stored in the initialized read-write area. And a global C statement like `const char string[] = "hello world"` would be stored in the initialized read-only area and the character pointer variable `string` in the initialized read-write area. Ex: `static int i = 10` will be stored in the data segment and `global int i = 10` will also be stored in data segment
- **3. Uninitialized Data Segment:** Uninitialized data segment often called the "**bss**" segment, named after an ancient assembler operator that stood for "**block started by symbol.**" Data in this segment is initialized by the compiler to arithmetic 0 before the program starts executing uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.
- For instance, a variable declared `static int i;` would be contained in the BSS segment. For instance, a global variable declared `int j;` would be contained in the BSS segment.
- **4. Stack:** The stack area traditionally adjoined the heap area and grew in the opposite direction; when the stack pointer met the heap pointer, free memory was exhausted. (With modern large address spaces and virtual memory techniques they may be placed almost anywhere, but they still typically grow in opposite directions.) The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. On the standard PC x86 computer architecture, it grows toward address zero; on some other architectures, it grows in the opposite direction. A "stack pointer" register tracks the top of the stack; it is adjusted each time a value is "pushed" onto the stack. The set of values pushed for one function call is termed a "stack frame"; A stack frame consists at minimum of a return address. Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function.
- **5. Heap:** Heap is the segment where dynamic memory allocation usually takes place.
- The heap area begins at the end of the BSS segment and grows to larger addresses from there. The Heap area is managed by `malloc`, `realloc`, and

free, which may use the brk and sbrk system calls to adjust its size (note that the use of brk/sbrk and a single “heap area” is not required to fulfill the contract of malloc/realloc/free; they may also be implemented using mmap to reserve potentially non-contiguous regions of virtual memory into the process’ virtual address space). The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

3. Examples:

3.1. The size(1) command reports the sizes (in bytes) of the text, data, and bss segments. (for more details please refer man page of size(1))

1. Check the following simple C program

```
#include <stdio.h>
int main(void)
{
    return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text      data      bss      dec      hex      filename
960       248        8      1216     4c0      memory-layout
```

3.2. Let us add one global variable in the program, now check the size of bss (highlighted in red color).

```
#include <stdio.h>
int global; /* Uninitialized variable stored in bss */
int main(void)
{
    return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text      data      bss      dec      hex      filename
960       248       12      1220     4c4      memory-layout
```

3.3. Let us add one static variable which is also stored in bss.

```
#include <stdio.h>
int global; /* Uninitialized variable stored in bss */
int main(void)
{
    static int i; /* Uninitialized static variable stored in bss */
    return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text      data      bss      dec      hex      filename
960       248       16      1224     4c8      memory-layout
```

3.4. Let us initialize the static variable which will then be stored in the Data Segment (DS)

```
#include <stdio.h>
int global; /* Uninitialized variable stored in bss*/
int main(void)
{
    static int i = 100; /* Initialized static variable stored in DS*/
    return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text      data      bss      dec      hex      filename
960       252       12      1224     4c8      memory-layout
```

3.5. Let us initialize the global variable which will then be stored in the Data Segment (DS)

```
#include <stdio.h>
int global = 10; /* initialized global variable stored in DS*/
int main(void)
{
    static int i = 100; /* Initialized static variable stored in DS*/
    return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text      data      bss      dec      hex      filename
960       256       8      1224     4c8      memory-layout
```

4. Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc()

Since C is a structured language, it has some fixed rules for programming. One of them includes changing the size of an array. An array is a collection of items stored at contiguous memory locations.

40	55	63	17	22	68	89	97	89
0	1	2	3	4	5	6	7	8

<- Array Indices

Array Length = 9

First Index = 0

Last Index = 8

As can be seen, the length (size) of the array above is 9. But what if there is a requirement to change this length (size)? For example,

- If there is a situation where only 5 elements are needed to be entered in this array. In this case, the remaining 4 indices are just wasting memory

in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.

- Take another situation. In this, there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case, 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.

This procedure is referred to as **Dynamic Memory Allocation in C**.

Dynamic memory allocation using `malloc()`, `calloc()`, `free()`, and `realloc()` is essential for efficient memory management in C. If you're looking to master these concepts and their application in data structures, the [C Programming Course Online with Data Structures](#) provides an in-depth understanding of memory allocation and management in C.

Therefore, C **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime. C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under `<stdlib.h>` header file to facilitate dynamic memory allocation in C programming. They are:

1. `malloc()`
2. `calloc()`
3. `free()`
4. `realloc()`

Let's look at each of them in greater detail.

4.1. C `malloc()` method

The "**malloc**" or "**memory allocation**" method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn't initialize memory at execution time so that it has initialized each block with the default garbage value initially.

Syntax of `malloc()` in C

```
ptr = (cast-type*) malloc(byte-size)
```

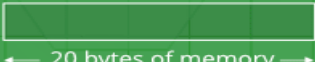
For Example:

```
ptr = (int*) malloc(100 * sizeof(int));
```

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.

Malloc()

```
int* ptr = ( int* ) malloc ( 5* sizeof ( int ) );
```

ptr = 

← 20 bytes of memory →

A large 20 bytes memory block is dynamically allocated to ptr

4 bytes



If space is insufficient, allocation fails and returns a NULL pointer.

4.2. Example of malloc() in C

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;
    // Get the number of elements for the array
    printf("Enter number of elements:");
    scanf("%d",&n);
    printf("Entered number of elements: %d\n", n);
    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));
    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL)
    {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");
        // Get the elements of the array
        for (i = 0; i < n; ++i)
        {
            ptr[i] = i + 1;
        }
        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }
    return 0;
}
```

```
}
```

Output

```
Enter number of elements:7
```

```
Entered number of elements: 7
```

```
Memory successfully allocated using malloc.
```

```
The elements of the array are: 1, 2, 3, 4, 5, 6, 7,
```

4.3. C calloc() method

1. “**calloc**” or “**contiguous allocation**” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:
2. It initializes each block with a default value ‘0’.
3. It has two parameters or arguments as compare to malloc().

Syntax of calloc() in C

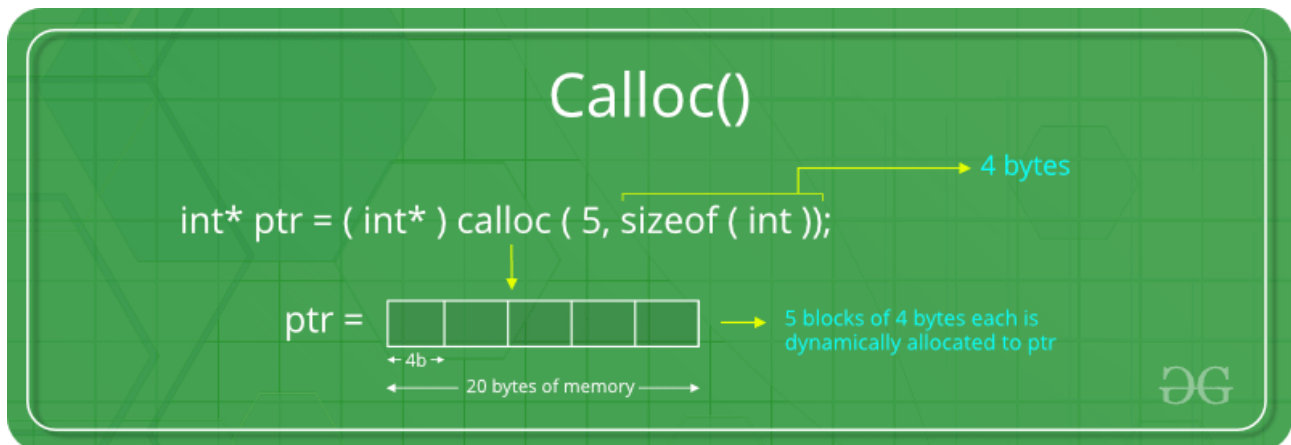
```
ptr = (cast-type*)calloc(n, element-size);
```

here, n is the no. of elements and element-size is the size of each element.

For Example:

```
ptr = (float*) calloc(25, sizeof(float));
```

This statement allocates contiguous space in memory for 25 elements each with the size of the float.



If space is insufficient, allocation fails and returns a NULL pointer.

4.4. Example of calloc() in C

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;
```

```

// Get the number of elements for the array
n = 5;
printf("Enter number of elements: %d\n", n);
// Dynamically allocate memory using calloc()
ptr = (int*)calloc(n, sizeof(int));
// Check if the memory has been successfully
// allocated by calloc or not
if (ptr == NULL) {
printf("Memory not allocated.\n");
exit(0);
}
else {
// Memory has been successfully allocated
printf("Memory successfully allocated using calloc.\n");
// Get the elements of the array
for (i = 0; i < n; ++i) {
    ptr[i] = i + 1;
}
// Print the elements of the array
printf("The elements of the array are: ");
for (i = 0; i < n; ++i) {
    printf("%d, ", ptr[i]);
}
}
return 0;
}

```

Output

```

Enter number of elements: 5
Memory successfully allocated using calloc.
The elements of the array are: 1, 2, 3, 4, 5,

```

4.5. C free() method

“**free**” method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax of free() in C

```
free(ptr);
```


Free()

```
int* ptr = ( int* ) calloc ( 5, sizeof ( int ) );
```



operation on ptr

free(ptr)



The memory of ptr is released



GG

4.6. Example of free() in C

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int main()
```

```
{
```

```
// This pointer will hold the
```

```
// base address of the block created
```

```
int *ptr, *ptr1;
```

```
int n, i;
```

```
// Get the number of elements for the array
```

```
n = 5;
```

```
printf("Enter number of elements: %d\n", n);
```

```
// Dynamically allocate memory using malloc()
```

```
ptr = (int*)malloc(n * sizeof(int));
```

```
// Dynamically allocate memory using calloc()
```

```
ptr1 = (int*)calloc(n, sizeof(int));
```

```
// Check if the memory has been successfully
```

```
// allocated by malloc or not
```

```
if (ptr == NULL || ptr1 == NULL) {
```

```
    printf("Memory not allocated.\n");
```

```
    exit(0);
```

```

}
else {
    // Memory has been successfully allocated
    printf("Memory successfully allocated using malloc.\n");
    // Free the memory
    free(ptr);
    printf("Malloc Memory successfully freed.\n");
    // Memory has been successfully allocated
    printf("\nMemory successfully allocated using calloc.\n");
    // Free the memory
    free(ptr1);
    printf("Calloc Memory successfully freed.\n");
}
return 0;
}

```

Output

```

Enter number of elements: 5
Memory successfully allocated using malloc.
Malloc Memory successfully freed.
Memory successfully allocated using calloc.
Calloc Memory successfully freed.

```

4.7. C realloc() method

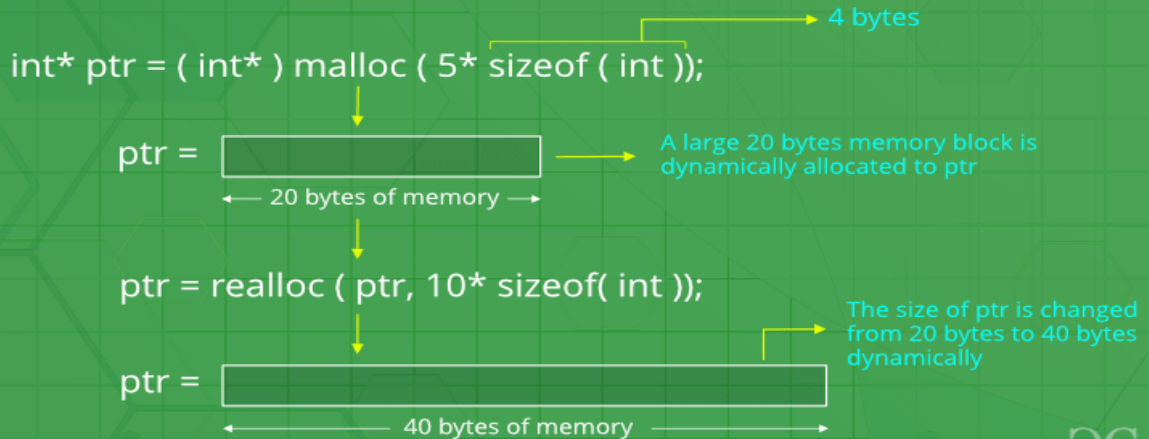
“**realloc**” or “**re-allocation**” method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

Syntax of realloc() in C

```
ptr = realloc(ptr, newSize);
```

where ptr is reallocated with new size 'newSize'.

Realloc()



If space is insufficient, allocation fails and returns a NULL pointer.

4.8. Example of `realloc()` in C

```
#include <stdio.h>

#include <stdlib.h>

int main()
{
    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);
    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));
    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
```

```

else {
    // Memory has been successfully allocated
    printf("Memory successfully allocated using calloc.\n");
    // Get the elements of the array
    for (i = 0; i < n; ++i) {
        ptr[i] = i + 1;
    }
    // Print the elements of the array
    printf("The elements of the array are: ");
    for (i = 0; i < n; ++i) {
        printf("%d, ", ptr[i]);
    }
    // Get the new size for the array
    n = 10;
    printf("\n\nEnter the new size of the array: %d\n", n);
    // Dynamically re-allocate memory using realloc()
    ptr = (int*)realloc(ptr, n * sizeof(int));
    if (ptr == NULL) {
        printf("Reallocation Failed\n");
        exit(0);
    }
    // Memory has been successfully allocated
    printf("Memory successfully re-allocated using realloc.\n");
    // Get the new elements of the array
    for (i = 5; i < n; ++i) {
        ptr[i] = i + 1;
    }
    // Print the elements of the array
    printf("The elements of the array are: ");
    for (i = 0; i < n; ++i) {

```

```

        printf("%d, ", ptr[i]);
    }
    free(ptr);
}
return 0;
}

```

Output

```

Enter number of elements: 5
Memory successfully allocated using calloc.
The elements of the array are: 1, 2, 3, 4, 5,
Enter the new size of the array: 10
Memory successfully re-allocated using realloc.
The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,

```

4.9. One another example for realloc() method is:

```

#include <stdio.h>

#include <stdlib.h>

int main()
{
    int index = 0, i = 0, n,*marks;    // this marks pointer hold the base address
                                       // of the block created

    int ans;

    marks = (int*)malloc(sizeof(
        int)); // dynamically allocate memory using malloc
    // check if the memory is successfully allocated by
    // malloc or not?
    if (marks == NULL) {
        printf("memory cannot be allocated");
    }
    else {
        // memory has successfully allocated
        printf("Memory has been successfully allocated by " "using malloc\n");
        printf("\n marks = %pc\n", marks); // print the base or beginning
                                           // address of allocated memory
    }
}

```

```

do {
    printf("\n Enter Marks\n");
    scanf("%d", &marks[index]); // Get the marks
    printf("would you like to add more(1/0): ");
    scanf("%d", &ans);
    if (ans == 1) {
        index++;
        marks = (int*)realloc(marks,(index + 1)* sizeof(int));
        // Dynamically reallocate
        // memory by using realloc
        // check if the memory is successfully
        // allocated by realloc or not?

        if (marks == NULL) {
            printf("memory cannot be allocated");
        }
        else {
            printf("Memory has been successfully " "reallocated using realloc:\n");
            printf("\n base address of marks are:%pc",marks); ////print the base or
            //beginning address of
            //allocated memory

        }
    }
} while (ans == 1); // print the marks of the students
for (i = 0; i <= index; i++) {
    printf("marks of students %d are: %d\n ", i, marks[i]);
}

free(marks);
}

return 0;
}

```

Output

```
Memory has been successfully allocated by using malloc

marks = 0x22eb010c

Enter Marks:89
would you like to add more(1/0): 1
Memory has been successfully reallocated using realloc:

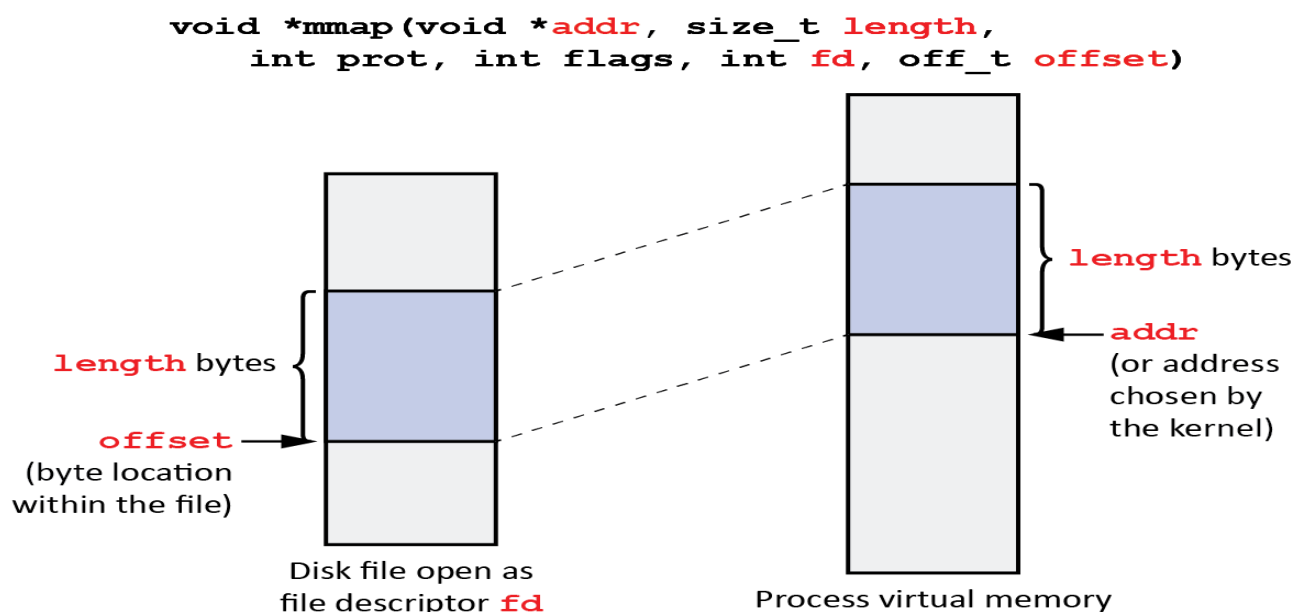
base address of marks are:0x22eb010c
Enter Marks:78
would you like to add more(1/0): 1
Memory has been successfully reallocated using realloc:

base address of marks are:0x22eb010c
Enter Marks:84
would you like to add more(1/0): 0
marks of students 0 are: 89
marks of students 1 are: 78
marks of students 2 are: 84
```

5. The `mmap()` System Call

`mmap()` is a system call that can be used by a user process to ask the operating system kernel to *map* either files or devices into the memory (i.e., address space) of that process. The `mmap()` system call can also be used to allocate memory (an anonymous mapping). A key point here is that the mapped pages are not actually brought into physical memory until they are referenced; thus `mmap()` can be used to implement lazy loading of pages into memory (demand paging).

The following figure illustrates the use of the `mmap()` system call:



Here is the function prototype for `mmap()`:

```
void *mmap(void *addr, size_t length, int prot, int flags,
int fd, off_t offset);
```

There are six arguments to the `mmap()` system call:

- `addr` - A hint to the operating system kernel as to the address at which the virtual mapping should start in the virtual memory (i.e., the virtual address space) of the process. The value can be specified as `NULL` to indicate that the kernel can place the virtual mapping anywhere it sees fit. If not `NULL`, then `addr` should be a multiple of the page size.
- `length` - The length (number of bytes) for the mapping.
- `prot` - The protection for the mapped memory. The value of `prot` is the bitwise or of various of the following single-bit values:
 - `PROT_READ` - Enable the contents of the mapped memory to be readable by the process.
 - `PROT_WRITE` - Enable the contents of the mapped memory to be writable by the process.
 - `PROT_EXEC` - Enable the contents of the mapped memory to be executable by the process as CPU machine instructions.
- `flags` - Various options controlling the mapping. Some of the more common `flags` values are described below:
 - `MAP_ANONYMOUS` (or `MAP_ANON`) - Allocate anonymous memory; the pages are not backed by any file.
 - `MAP_FILE` - The default setting; it need not be specified. The mapped region is backed by a regular file.
 - `MAP_FIXED` - Don't interpret `addr` as a hint: place the mapping at exactly that address, which must be a multiple of the page size.
 - `MAP_PRIVATE` - Modifications to the mapped memory region are not visible to other processes mapping the same file.
 - `MAP_SHARED` - Modifications to the mapped memory region are visible to other processes mapping the same file and are eventually reflected in the file.
- `fd` - The open file descriptor for the file from which to populate the memory region. If `MAP_ANONYMOUS` is specified, then `fd` should be given as `-1`.
- `offset` - If this is not an anonymous mapping, the memory mapped region will be populated with data starting at position `offset` bytes from the beginning of the file open as file descriptor `fd`. Should be a multiple of the page size.

On success, `mmap()` returns a pointer to the mapped area. On error, the value `MAP_FAILED` (that is, `(void *) (-1)`) is returned and `errno` is set to indicate the reason. Full details on the `mmap()` system call are available using the `man mmap` command.

The munmap () System Call

`munmap ()` is a system call used to unmap memory previously mapped with `mmap ()`. The call removes the mapping for the memory from the address space of the calling process process:

```
int munmap(void *addr, size_t length);
```

There are two arguments to the `munmap ()` system call:

- `addr` - The address of the memory to unmap from the calling process's virtual mapping. Should be a multiple of the page size.
- `length` - The length of the memory (number of bytes) to unmap from the calling process's virtual mapping. Should be a multiple of the page size.

On success, `munmap ()` returns 0, on failure -1 and `errno` is set to indicate the reason. If successful, future accesses to the unmapped memory area will result in a segmentation fault (SIGSEGV).

6. Advantages of Using mmap()

There are many advantages to using `mmap ()` to gain access to the contents of some file on disk.

One advantage of using `mmap ()` is **lazy loading**. If no memory within a certain page is ever referenced, then that page is never loaded into physical memory. This can be crucial in certain applications in terms of saving both memory and time.

Another advantage with `mmap ()` is **speed improvements**. Traditional I/O involves a lot of system calls (e.g., calls to `read ()`) to load data into memory. There are costs associated with these calls, such as error checking within the functions themselves. Loading data into main memory also has to go through several layers of software abstraction, with the data being copied around in various buffers in the operating system before finally being placed in memory, which clearly will slow down the program. Using `mmap ()` avoids both of these issues.

Finally, `mmap ()` has the advantage of being able to support **versatile dynamic memory allocation**. In particular, `malloc ()` cannot safely be called in a signal handler, whereas `mmap ()` can, using anonymous mappings. Additionally, `malloc ()` itself can be made more versatile by using `mmap ()` to allocate memory rather than simply raising the process's break by using `sbrk ()`. Calling `sbrk ()` necessitates that memory may be freed only at the top of the heap (i.e., by lowering the break). However, there is no such restriction if memory is allocated using `mmap ()`. As such, memory in the middle of the heap may be freed at will (using `munmap ()`).

7. An mmap () Example

The following simple, but buggy, example program appears in your lab repo as `exercisel.c`. This program demonstrates a typical use case of `mmap()` for reading data from a file (the `#include` lines in this example have been omitted here for simplicity of presentation).

```
/*
 *Requires:
 *This program's source code must reside in the current
working directory
 *in a readable file named "exercisel.c".
 *
 *Effects:
 *Reads this program's source code, via mmap(), and prints
that source
 *code to stdout.
 */
int
main(void)
{
    struct stat stat;
    int fd, size;
    char *buf;
    // Open the file and get its size.
    fd = open("exercisel.c", O_RDONLY);
    if (fd < 0 || fstat(fd, &stat) < 0) {
        fprintf(stderr, "open() or fstat() failed.\n");
        return (1);
    }
    size = stat.st_size;
    // Map the file to a new virtual memory area.
    buf = mmap(NULL, size, PROT_NONE, MAP_PRIVATE, fd, 0);
    if (buf == MAP_FAILED) {
        fprintf(stderr, "mmap() failed.\n");
        return (1);
    }
    // Print out the contents of the file to stdout.
    for (int i = 0; i < size; i++) {
        printf("%c", buf[i]);
    }
    // Clean up. Ignore the return values because we exit anyway.
    (void)munmap(buf, size);
    (void)close(fd);
    return (0);
}
```

This program calls the `open()` system call to open its own source code file (the file `exercisel.c`) and then gets the size (in bytes) of this file by calling the `fstat()` system call. The `fstat()` system call returns a number of different pieces of information about the status of the file open on the given file descriptor.

(The `stat()` system call is identical to `fstat()`, except that `stat()` takes the name of the file as its first argument rather than a file descriptor open to that file, as in `fstat()`.) The program then uses `mmap()` to map that file into the process's memory, and then prints out the contents of the file from the memory into which the file has been mapped.

8. C Memory Management Example

To demonstrate a practical example of **dynamic memory**, we created a program that can make a list of any length.

Regular arrays in C have a fixed length and cannot be changed, but with dynamic memory we can create a list as long as we like:

```
struct list {
    int *data;

    // Points to the memory where the list items are stored
    int numItems;

    // Indicates how many items are currently in the list
    int size;

    // Indicates how many items fit in the allocated memory
};

void addToList(struct list *myList, int item);
int main() {
    struct list myList;
    int amount;

    // Create a list and start with enough space for 10 items
    myList.numItems = 0;
    myList.size = 10;
    myList.data = malloc(myList.size * sizeof(int));
    // Find out if memory allocation was successful
    if (myList.data == NULL) {
        printf("Memory allocation failed");
        return 1; // Exit the program with an error code
    }

    // Add any number of items to the list specified by the amount
    // variable
    amount = 44;
    for (int i = 0; i < amount; i++) {
        addToList(&myList, i + 1);
    }

    // Display the contents of the list
    for (int j = 0; j < myList.numItems; j++) {
        printf("%d ", myList.data[j]);
    }
}
```

```

    // Free the memory when it is no longer needed
    free(myList.data);
    myList.data = NULL;
    return 0;
}
// This function adds an item to a list
void addToList(struct list *myList, int item) {
    // If the list is full then resize the memory to fit 10 more items
    if (myList->numItems == myList->size) {
        myList->size += 10;
        myList->data = realloc(myList->data, myList->size*sizeof(int)
    );
    }
    // Add the item to the end of the list
    myList->data[myList->numItems] = item;
    myList->numItems++;
}

```

Example explained

This example has three parts:

- A structure `myList` that contains a list's data
- The `main()` function with the program in it.
- A function `addToList()` which adds an item to the list

The `myList` structure

The `myList` structure contains all of the information about the list, including its contents. It has three members:

- `data` - A pointer to the dynamic memory which contains the contents of the list
- `numItems` - Indicates the number of items that list has
- `size` - Indicates how many items can fit in the allocated memory

We use a structure so that we can easily pass all of this information into a function.

The `main()` function

The `main()` function starts by initializing the list with space for 10 items:

```

// Create a list and start with enough space for 10 items
myList.numItems = 0;

```

```
myList.size = 10;
myList.data = malloc(myList.size * sizeof(int));
```

`myList.numItems` is set to 0 because the list starts off empty.

`myList.size` keeps track of how much memory is reserved. We set it to 10 because we will reserve enough memory for 10 items.

We then allocate the memory and store a pointer to it in `myList.data`.

Then we include error checking to find out if memory allocation was successful:

```
// Find out if memory allocation was successful
if (myList.data == NULL) {
    printf("Memory allocation failed");
    return 1; // Exit the program with an error code
}
```

If everything is fine, a loop adds 44 items to the list using the `addToList()` function:

```
// Add any number of items to the list specified by the amount
// variable
amount = 44;
for (int i = 0; i < amount; i++) {
    addToList(&myList, i + 1);
}
```

In the code above, `&myList` is a pointer to the list and `i + 1` is a number that we want to add to the list. We chose `i + 1` so that the list would start at 1 instead of 0. You can choose any number to add to the list.

After all of the items have been added to the list, the next loop prints the contents of the list.

```
// Display the contents of the list
for (int j = 0; j < myList.numItems; j++) {
    printf("%d ", myList.data[j]);
}
```

When we finish printing the list we free the memory to prevent memory leaks.

```
// Free the memory when it is no longer needed
free(myList.data);
myList.data = NULL;
```

The `addToList()` function

Our `addToList()` function adds an item to the list. It takes two parameters:

```
void addToList(struct list *myList, int item)
```

1. A pointer to the list.
2. The value to be added to the list.

The function first checks if the list is full by comparing the number of items in the list to the size of the list. If the list is full then it reallocates the memory to fit 10 more items:

```
// If the list is full then resize the memory to fit 10 more items
if (myList->numItems == myList->size) {
    myList->size += 10;
    myList->data = realloc( myList->data, myList->size * sizeof(int)
);
}
```

Finally, the function adds the item to the end of list. The index at `myList->numItems` is always at the end of the list because it increases by 1 each time a new item is added.

```
// Add the item to the end of the list
myList->data[myList->numItems] = item;
myList->numItems++;
```

Why do we reserve 10 items at a time?

Optimizing is a balancing act between memory and performance. Even though we may be allocating some memory that we are not using, reallocating memory too frequently can be inefficient. There is a balance between allocating too much memory and allocating memory too frequently.

We chose the number 10 for this example, but it depends on how much data you expect and how often it changes. For example, if we know beforehand that we are going to have exactly 44 items then we can allocate memory for exactly 44 items and only allocate it once.

9. Real-Life Memory Management Example

10.