| Experiment # | <TO BE FILLED BY STUDENT> | Student ID | <TO BE FILLED BY STUDENT> |
|---|---|---|---|
| Date | <TO BE FILLED BY STUDENT> | Student Name | [@KLWKS_BOT THANOS] |

**Experiment Title: Memory Allocation Techniques**

**Aim/Objective:**

Efficiently allocate and manage memory resources to optimize system performance and facilitate the execution of processes and applications.

**Description:**

Memory allocation techniques in operating systems refer to the methods used to allocate and manage memory resources for processes and applications. These techniques aim to optimize memory utilization, ensure efficient allocation, and provide a fair and balanced distribution of memory among processes.

**Pre-Requisites:**

- General idea on memory allocation techniques
- malloc(), realloc(), calloc(), free() library functions
- Concept of altering program break
- Basic strategies for managing free space (first-fit, best-fit, worst-fit)
- Concepts of Splitting and coalescing in free space management

**Pre-Lab:**

| **Memory Allocation Techniques** | **FUNCTIONALITY** |
|---|---|
| Virtual Memory | Allows using disk space as extended memory for processes. |
| Swapping | Transfers processes between main memory and disk for optimization. |
| Paging | Divides memory into fixed-size pages for non-contiguous allocation. |

| | |
|---|---|
| Fragmentation | Occurs when memory becomes inefficient due to allocation/deallocation. |
| Segmentation | Divides memory into variable-sized segments based on process requirements. |
| Internal Fragmentation | Wasted memory within allocated blocks due to fixed sizes. |
| External Fragmentation | Unused small spaces between allocated memory blocks in main memory. |
| Free Space Management | Tracks available memory blocks for efficient allocation and deallocation. |
| Splitting | Divides larger memory blocks into smaller blocks to satisfy requests. |
| Coalescing | Merges adjacent free blocks to reduce fragmentation and optimize space. |

**In-Lab:**

1. write a C program to implement Dynamic Memory Allocation.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   int n;
   printf("Enter the size of the array: ");
   scanf("%d", &n);
   int *arr = (int *)malloc(n * sizeof(int));
   if (arr == NULL) {
      printf("Memory allocation failed. Exiting...\n");
      return 1;
   }
   printf("Enter %d integers:\n", n);
   for (int i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }
   printf("The array contains: ");
   for (int i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }
```

```
    printf("\n");

      free(arr);

       return 0;

    }
```

2. write a C program to implement the Memory Management concept using the technique of Best fitalgorithms.

```
#include <stdio.h>
#include <stdlib.h>
#define MEMORY_SIZE 100

typedef struct MemoryBlock {
   int size;
   int allocated;
} MemoryBlock;

MemoryBlock memory[MEMORY_SIZE];

void initializeMemory() {
   for (int i = 0; i < MEMORY_SIZE; i++) {
      memory[i].size = 0;
      memory[i].allocated = 0;
   }
}

void allocateBestFit(int blockSize) {
   int bestFitIndex = -1;
   int bestFitSize = MEMORY_SIZE + 1;
   for (int i = 0; i < MEMORY_SIZE; i++) {
      if (!memory[i].allocated && memory[i].size >= blockSize) {
         if (memory[i].size < bestFitSize) {
            bestFitSize = memory[i].size;
            bestFitIndex = i;
         }
      }
   }
   if (bestFitIndex != -1) {
      memory[bestFitIndex].allocated = 1;
```

```
    printf("Allocated %d bytes at index %d.\n", blockSize, bestFitIndex);

    } else {
        printf("No suitable block found for allocation.\n");
    }
}

void deallocate(int blockIndex) {
    if (blockIndex >= 0 && blockIndex < MEMORY_SIZE &&
memory[blockIndex].allocated) {
        memory[blockIndex].allocated = 0;
        printf("Deallocated %d bytes at index %d.\n", memory[blockIndex].size,
blockIndex);
    } else {
        printf("Invalid deallocation request.\n");
    }
}

int main() {
    initializeMemory();

    allocateBestFit(20);
    allocateBestFit(30);
    allocateBestFit(15);

    deallocate(1);

    return 0;
}
```

3. write a C program to implement the Memory Management concept using the technique worst fitalgorithms.

```
#include <stdio.h>
#include <stdlib.h>
#define MEMORY_SIZE 100

typedef struct MemoryBlock {
   int size;
   int allocated;
} MemoryBlock;

MemoryBlock memory[MEMORY_SIZE];

void initializeMemory() {
   for (int i = 0; i < MEMORY_SIZE; i++) {
      memory[i].size = 0;
      memory[i].allocated = 0;
   }
}

void allocateWorstFit(int blockSize) {
   int worstFitIndex = -1;
   int worstFitSize = -1;
   for (int i = 0; i < MEMORY_SIZE; i++) {
      if (!memory[i].allocated && memory[i].size >= blockSize) {
         if (memory[i].size > worstFitSize) {
            worstFitSize = memory[i].size;
            worstFitIndex = i;
         }
      }
   }
   if (worstFitIndex != -1) {
      memory[worstFitIndex].allocated = 1;
      printf("Allocated %d bytes at index %d.\n", blockSize, worstFitIndex);
   } else {
      printf("No suitable block found for allocation.\n");
   }
}
```

```
    void deallocate(int blockIndex) {
  if (blockIndex >= 0 && blockIndex < MEMORY_SIZE && memory[blockIndex].allocated) {

    memory[blockIndex].allocated = 0;
     printf("Deallocated %d bytes at index %d.\n", memory[blockIndex].size, blockIndex);
  } else {
     printf("Invalid deallocation request.\n");
  }
}

int main() {
  initializeMemory();

  allocateWorstFit(20);
  allocateWorstFit(30);
  allocateWorstFit(15);

  deallocate(1);

  return 0;
}
```

4. write a C program to implement the Memory Management concept using the technique of first fitalgorithms.

```
#include <stdio.h>
#include <stdlib.h>
#define MEMORY_SIZE 100
typedef struct MemoryBlock {
   int size;
   int allocated;
} MemoryBlock;
MemoryBlock memory[MEMORY_SIZE];

void initializeMemory() {
   for (int i = 0; i < MEMORY_SIZE; i++) {
      memory[i].size = 0;
      memory[i].allocated = 0;
   }
}

void allocateFirstFit(int blockSize) {
   int firstFitIndex = -1;
   for (int i = 0; i < MEMORY_SIZE; i++) {
      if (!memory[i].allocated && memory[i].size >= blockSize) {
         firstFitIndex = i;
         break;
      }
   }
   if (firstFitIndex != -1) {
      memory[firstFitIndex].allocated = 1;
      printf("Allocated %d bytes at index %d.\n", blockSize, firstFitIndex);
   } else {
      printf("No suitable block found for allocation.\n");
   }
}
```

```
void deallocate(int blockIndex) {
    if (blockIndex >= 0 && blockIndex < MEMORY_SIZE && memory[blockIndex].allocated) {
        memory[blockIndex].allocated = 0;
        printf("Deallocated %d bytes at index %d.\n", memory[blockIndex].size, blockIndex);
    } else {
        printf("Invalid deallocation request.\n");
    }
}

int main() {
    initializeMemory();
    allocateFirstFit(20);
    allocateFirstFit(30);
    allocateFirstFit(15);
    deallocate(1);
    return 0;
}
```

5. write a C program to implement the paging concept for memory management.

```
#include <stdio.h>
#define PAGE_SIZE 4096
#define MEMORY_SIZE 65536
#define NUM_PAGES (MEMORY_SIZE / PAGE_SIZE)
#define NUM_FRAMES 16

int page_table[NUM_PAGES];
char memory[MEMORY_SIZE];
char disk[MEMORY_SIZE];

void initializeMemory() {
   for (int i = 0; i < NUM_PAGES; i++) {
      page_table[i] = -1;
   }

   for (int i = 0; i < MEMORY_SIZE; i++) {
      memory[i] = 'A' + (i % 26);
      disk[i] = memory[i];
   }
}

void loadPage(int pageNumber, int frameNumber) {
   int start = pageNumber * PAGE_SIZE;
   int frameStart = frameNumber * PAGE_SIZE;

   for (int i = 0; i < PAGE_SIZE; i++) {
      memory[frameStart + i] = disk[start + i];
   }
   page_table[pageNumber] = frameNumber;
}

char readMemory(int address) {
   int pageNumber = address / PAGE_SIZE;
   int offset = address % PAGE_SIZE;
   int frameNumber = page_table[pageNumber];

   if (frameNumber == -1) {
      printf("Page fault! Page %d is not in memory.\n", pageNumber);
```

```
 return 0;
   }
   int frameStart = frameNumber * PAGE_SIZE;
   return memory[frameStart + offset];
}

int main() {
   initializeMemory();
   int address = 8192;
   char data = readMemory(address);
   printf("Data at address %d: %c\n", address, data);
   loadPage(2, 0);
   data = readMemory(address);
   printf("Data at address %d after loading page: %c\n", address, data);
   return 0;
}
```

Data and Results (Program 1-3 ):

Program 1 :

**Data:**

Enter the size of the array, followed by integers to store in dynamically allocated memory.

**Result:**

The program successfully allocates, stores, and prints the entered integers from dynamically allocated memory.

Program 2 :

**Data**

The program demonstrates memory allocation using the Best Fit algorithm with memory block sizes.

**Result**

Memory allocated and deallocated successfully, displaying block allocation and deallocation statuses.

Program 3 :

**Data:** Memory management using Worst Fit allocation, handling allocation and deallocation for memory blocks.

**Result:** Allocated and deallocated memory blocks based on Worst Fit strategy.

Analysis and inferences:

**Analysis:**

The programs demonstrate memory allocation and deallocation strategies (Best Fit, Worst Fit, First Fit).

**Inferences:**

Memory allocation strategies impact performance and efficiency based on block sizes.

**Post-Lab:**

1. Demonstrate Allocating Memory on the Heap with your implementation of malloc() and free() using free list and UNIX system calls.

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

#define HEAP_SIZE 65536
#define ALIGNMENT 16

typedef struct Block {
   size_t size;
   struct Block* next;
} Block;

static Block* free_list = NULL;

void* malloc(size_t size) {
   if (size == 0) {
      return NULL;
   }
   size = (size + ALIGNMENT - 1) & ~(ALIGNMENT - 1);

   if (!free_list) {
      free_list = sbrk(0);
      if (sbrk(HEAP_SIZE) == (void*)-1) {
         return NULL;
      }
      free_list->size = HEAP_SIZE;
      free_list->next = NULL;
   }
```

```
    Block* prev = NULL;
    Block* current = free_list;

    while (current) {
        if (current->size >= size) {
            if (current->size > size + sizeof(Block)) {
                Block* new_block = (Block*)((char*)current + size);
                new_block->size = current->size - size;
                new_block->next = current->next;
                current->size = size;
                current->next = new_block;
            }
            if (prev) {
                prev->next = current->next;
            } else {
                free_list = current->next;
            }
            return (void*)(current + 1);
        }
        prev = current;
        current = current->next;
    }
    return NULL;
}

void free(void* ptr) {
    if (!ptr) {
        return;
    }

    Block* block = (Block*)ptr - 1;
    block->next = free_list;
    free_list = block;
```

```
}

void test_malloc_free() {
   printf("Allocating 1024 bytes...\n");
   void* ptr1 = malloc(1024);
   if (ptr1) {
      printf("Allocated 1024 bytes at address %p\n", ptr1);
   } else {
      printf("Memory allocation failed\n");
   }

   printf("Allocating 2048 bytes...\n");
   void* ptr2 = malloc(2048);
   if (ptr2) {
      printf("Allocated 2048 bytes at address %p\n", ptr2);
   } else {
      printf("Memory allocation failed\n");
   }

   printf("Deallocating the first block...\n");
   free(ptr1);

   printf("Allocating 512 bytes...\n");
   void* ptr3 = malloc(512);
   if (ptr3) {
      printf("Allocated 512 bytes at address %p\n", ptr3);
   } else {
      printf("Memory allocation failed\n");
   }

   printf("Deallocating all blocks...\n");
   free(ptr2);
   free(ptr3);
```

```
}

int main() {
   test_malloc_free();
   return 0;
}
```

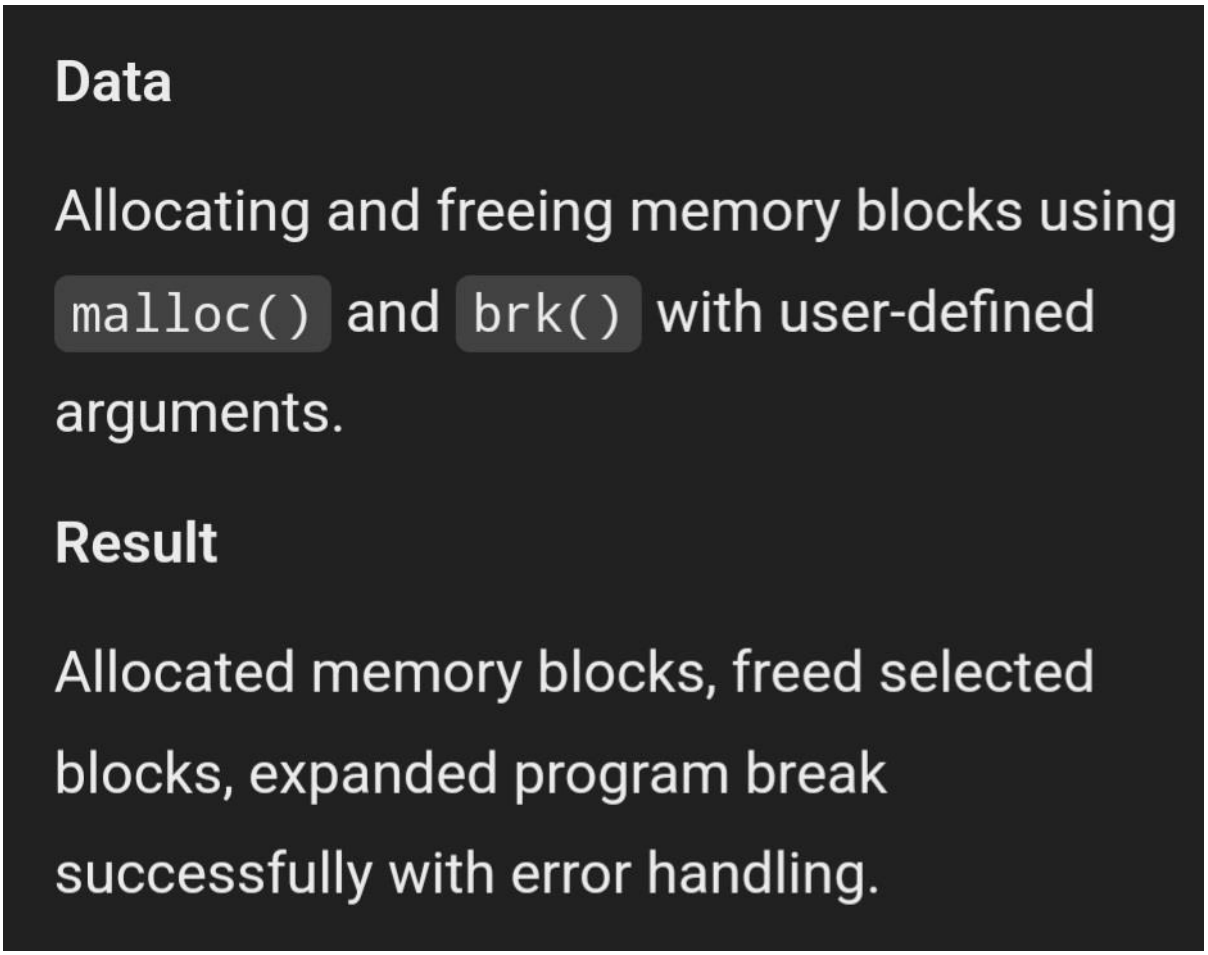
2. Develop a program to illustrate the effect of free() on the program break. This program allocates multiple blocks of memory and then frees some or all of them, depending on its (optional) command-line arguments. The first two command-line arguments specify the number and size of blocks to allocate. The third command-line argument specifies the loop step unit to be used when freeing memory blocks. If we specify 1 here (which is also the default if this argument is omitted), then the program frees every memory block; if 2, then every second allocated block; and so on. The fourth and fifth command-line arguments specify the range of blocks that we wish to free. If these arguments are omitted, then all allocated blocks (in steps given by the third command-line argument) are freed. Find the present address of the program break using sbrk() and expand the program break by the size 1000000 using brk().

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

int main(int argc, char *argv[]) {
   if (argc < 3) {
      printf("Usage: %s <num_blocks> <block_size> [step] [start] [end]\n", argv[0]);
      return 1;
   }

   int num_blocks = atoi(argv[1]);
   int block_size = atoi(argv[2]);
   int step = 1;
   int start = 0;
   int end = num_blocks;

   if (argc > 3) {
      step = atoi(argv[3]);
   }
   if (argc > 4) {
      start = atoi(argv[4]);
   }
   if (argc > 5) {
      end = atoi(argv[5]);
```

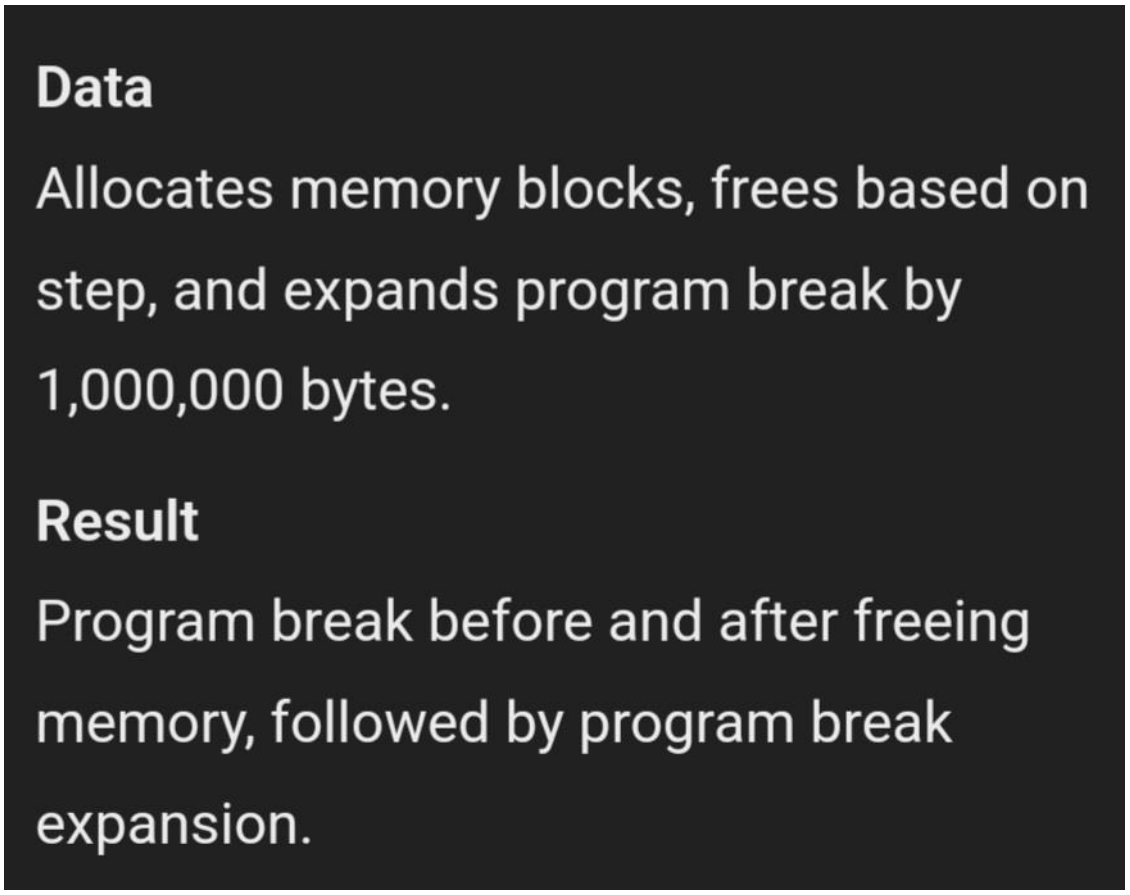
```
}

  if (num_blocks <= 0 || block_size <= 0 || step <= 0 || start < 0 || end < start || end > num_blocks) {
    printf("Invalid arguments.\n");
    return 1;
  }

  void **blocks = (void **)malloc(num_blocks * sizeof(void *));
  if (blocks == NULL) {
    perror("malloc");
    return 1;
  }

  printf("Allocating %d blocks of size %d bytes each.\n", num_blocks, block_size);
  for (int i = 0; i < num_blocks; i++) {
    blocks[i] = malloc(block_size);
    if (blocks[i] == NULL) {
      perror("malloc");
      return 1;
    }
  }

  printf("Program break before freeing blocks: %p\n", sbrk(0));
  printf("Press Enter to continue and free memory blocks...\n");
  getchar();

  printf("Freeing memory blocks from %d to %d with a step of %d.\n", start, end, step);
  for (int i = start; i < end; i += step) {
    free(blocks[i]);
    blocks[i] = NULL;
  }
```

```
  printf("Program break after freeing some blocks: %p\n", sbrk(0));

   printf("Expanding the program break by 1,000,000 bytes.\n");
   if (brk(sbrk(0) + 1000000) == 0) {
      printf("Program break expanded by 1,000,000 bytes.\n");
   } else {
      perror("brk");
   }

   free(blocks);
   return 0;
}
```

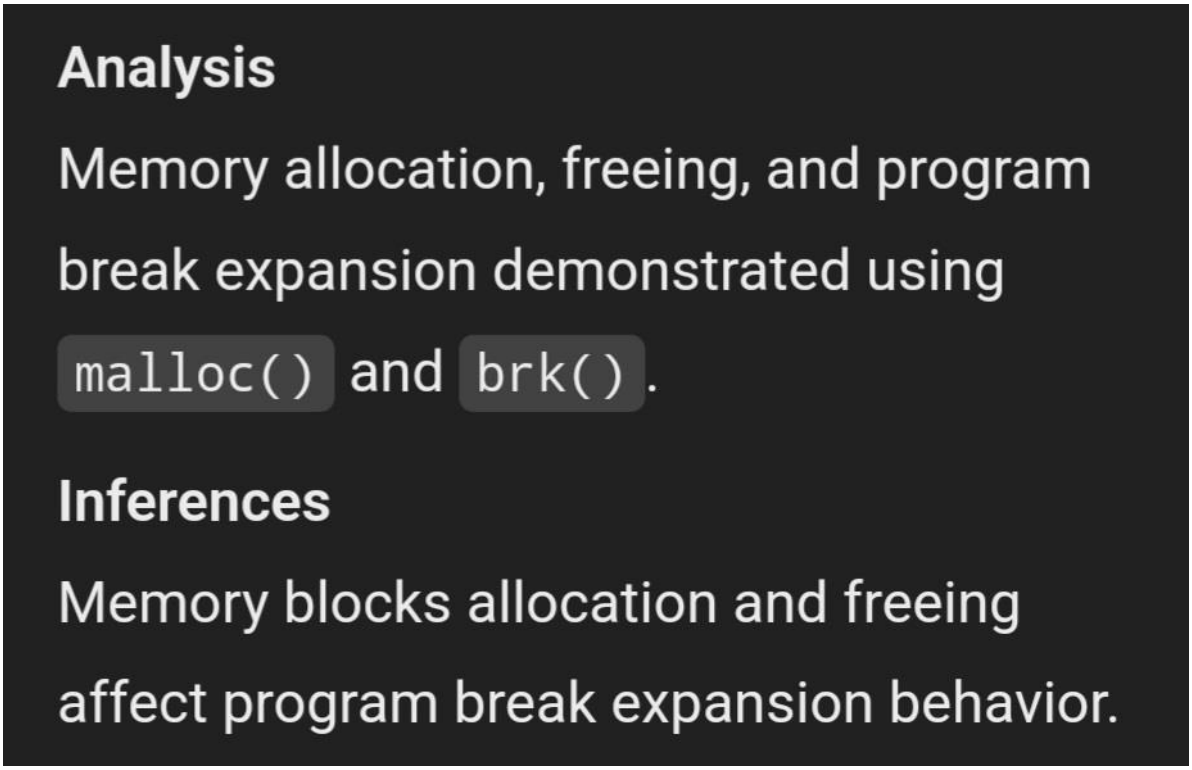
Data and Results (Program 7):



Data and Results (Program 8):

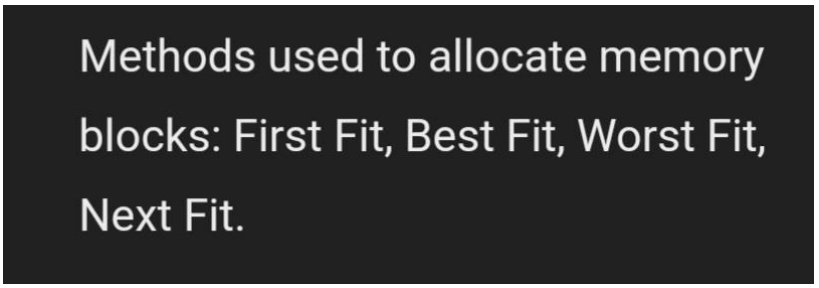


Analysis and Inferences:

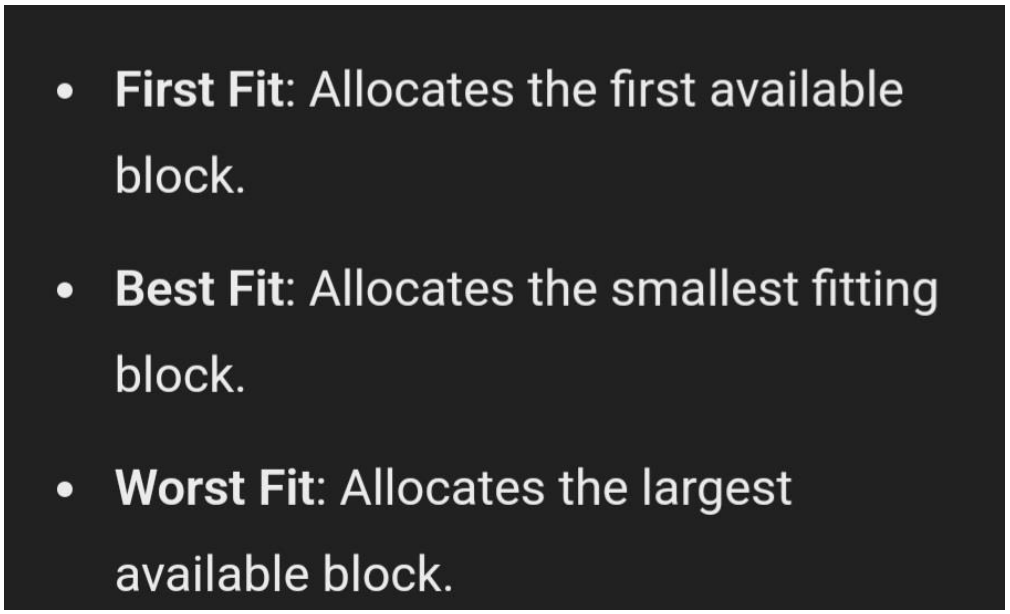


**Sample VIVA-VOCE Questions (In-Lab):**

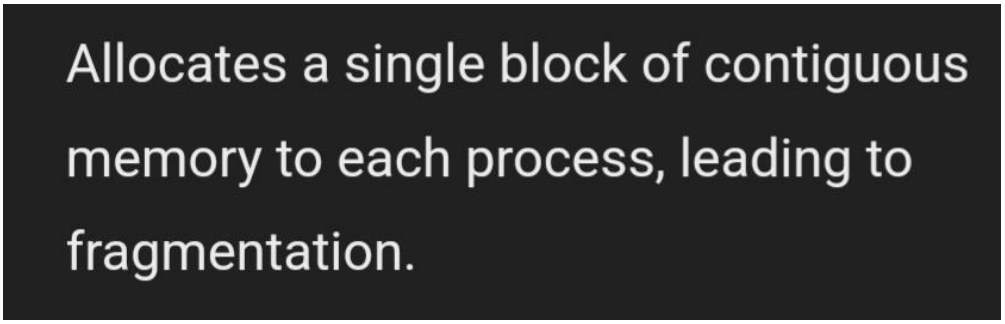
1. What are the Memory Allocation Strategies?



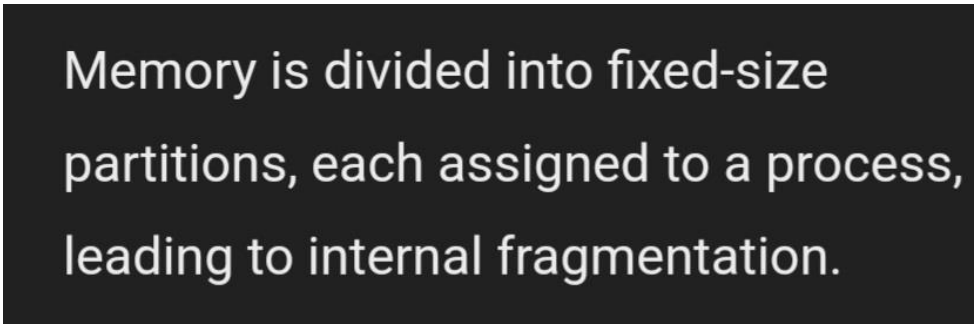

2. Differentiate between Best Fit, First Fit, and Worst Fit Strategies.



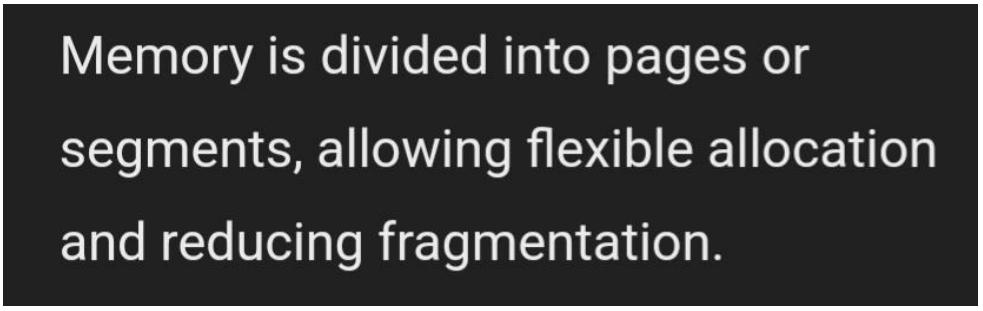

3. Explain in detail Contiguous Memory Allocation.



4. Explain in detail the Fixed Partition Allocation.




5. Explain in detail Non-Contiguous Memory Allocation.




| **Evaluator Remark (if any):** | **Marks Secured _____out of 50** |
|---|---|
| | **Signature of the Evaluator with Date** |

**Note: Evaluator MUST ask Viva-voce before signing and posting marks for each experiment.**