Assignment No 5:

Name: Samruddhi Devram Khilari

Branch & Year: AIML-B SY

Prn: 12420020

Roll no: 2

Sub: Operating System Lab

Aim: - Memory Management Technique

Q1] Implement Replacement Startegies: First Fit, Next fit, Bext fit and Worst fit.

```
#include <stdio.h>

#define MAX_MEMORY 100
#define MAX_PROCESSES 10

int memory[MAX_MEMORY]; // Simulating the memory
int processSizes[MAX_PROCESSES]; // Sizes of the processes
int numProcesses;

// Function to initialize memory
void initializeMemory() {
    for (int i = 0; i < MAX_MEMORY; i++) {
        memory[i] = -1; // -1 indicates free memory
    }
}

// Function to display memory state
void displayMemory() {
    printf("Memory Allocation: ");
    for (int i = 0; i < MAX_MEMORY; i++) {
        if (memory[i] != -1) {
            printf("[%d] ", memory[i]); // Process ID
        } else {
            printf("[ ] "); // Free space
        }
    }
    printf("\n");
}

// First Fit Strategy
void firstFit() {
    for (int i = 0; i < numProcesses; i++) {
        int j;
        for (j = 0; j < MAX_MEMORY; j++) {
            if (memory[j] == -1) { // Check for free memory
                int k;
                for (k = 0; k < processSizes[i]; k++) {
                    if (j + k >= MAX_MEMORY || memory[j + k] != -1) {
```

```
                break; // Not enough space
            }
        }
        if (k == processSizes[i]) {
            // Allocate memory
            for (k = 0; k < processSizes[i]; k++) {
                memory[j + k] = i; // Assign process ID
            }
            printf("First Fit: Process %d allocated starting at index %d\n", i, j);
            break;
        }
    }
  }
}

// Best Fit Strategy
void bestFit() {
  for (int i = 0; i < numProcesses; i++) {
    int bestIndex = -1;
    int bestSize = MAX_MEMORY + 1;

    for (int j = 0; j < MAX_MEMORY; j++) {
      if (memory[j] == -1) {
        int k;
        for (k = 0; k < processSizes[i]; k++) {
          if (j + k >= MAX_MEMORY || memory[j + k] != -1) {
            break; // Not enough space
          }
        }
        if (k == processSizes[i]) {
          int freeSize = 0;
          while (j + freeSize < MAX_MEMORY && memory[j + freeSize] == -1) {
            freeSize++;
          }
          if (freeSize < bestSize) {
            bestSize = freeSize;
            bestIndex = j;
          }
        }
      }
    }

    if (bestIndex != -1) {
      for (int k = 0; k < processSizes[i]; k++) {
        memory[bestIndex + k] = i; // Assign process ID
      }
      printf("Best Fit: Process %d allocated starting at index %d\n", i, bestIndex);
    }
  }
}

// Next Fit Strategy
void nextFit(int *lastIndex) {
```

```
    for (int i = 0; i < numProcesses; i++) {
        int j = *lastIndex;

        for (int k = 0; k < MAX_MEMORY; k++) {
            int idx = (j + k) % MAX_MEMORY; // Wrap around
            if (memory[idx] == -1) { // Check for free memory
                int l;
                for (l = 0; l < processSizes[i]; l++) {
                    if (idx + l >= MAX_MEMORY || memory[idx + l] != -1) {
                        break; // Not enough space
                    }
                }
                if (l == processSizes[i]) {
                    // Allocate memory
                    for (l = 0; l < processSizes[i]; l++) {
                        memory[idx + l] = i; // Assign process ID
                    }
                    printf("Next Fit: Process %d allocated starting at index %d\n", i, idx);
                    *lastIndex = (idx + l) % MAX_MEMORY; // Update last index
                    break;
                }
            }
        }
    }
}

// Worst Fit Strategy
void worstFit() {
    for (int i = 0; i < numProcesses; i++) {
        int worstIndex = -1;
        int worstSize = -1;

        for (int j = 0; j < MAX_MEMORY; j++) {
            if (memory[j] == -1) {
                int k;
                for (k = 0; k < processSizes[i]; k++) {
                    if (j + k >= MAX_MEMORY || memory[j + k] != -1) {
                        break; // Not enough space
                    }
                }
                if (k == processSizes[i]) {
                    int freeSize = 0;
                    while (j + freeSize < MAX_MEMORY && memory[j + freeSize] == -1) {
                        freeSize++;
                    }
                    if (freeSize > worstSize) {
                        worstSize = freeSize;
                        worstIndex = j;
                    }
                }
            }
        }

        if (worstIndex != -1) {
```

```
            for (int k = 0; k < processSizes[i]; k++) {
                memory[worstIndex + k] = i; // Assign process ID
            }
            printf("Worst Fit: Process %d allocated starting at index %d\n", i, worstIndex);
        }
    }
}

int main() {
    int lastIndex = 0;

    // Initialize memory
    initializeMemory();

    // Input process sizes
    printf("Enter the number of processes (max %d): ", MAX_PROCESSES);
    scanf("%d", &numProcesses);
    printf("Enter the sizes of the processes:\n");
    for (int i = 0; i < numProcesses; i++) {
        printf("Process %d size: ", i);
        scanf("%d", &processSizes[i]);
    }

    // Perform allocations
    printf("\n--- First Fit ---\n");
    firstFit();
    displayMemory();

    // Reset memory for next allocation
    initializeMemory();

    printf("\n--- Best Fit ---\n");
    bestFit();
    displayMemory();

    // Reset memory for next allocation
    initializeMemory();

    printf("\n--- Next Fit ---\n");
    nextFit(&lastIndex);
    displayMemory();

    // Reset memory for next allocation
    initializeMemory();

    printf("\n--- Worst Fit ---\n");
    worstFit();
    displayMemory();

    return 0;
}
```

Output ===

```
Enter the number of processes (max 10): 5
Enter the sizes of the processes:
Process 0 size: 15
Process 1 size: 10
Process 2 size: 20
Process 3 size: 8
Process 4 size: 4

--- First Fit ---
First Fit: Process 0 allocated starting at index 0
First Fit: Process 1 allocated starting at index 15
First Fit: Process 2 allocated starting at index 25
First Fit: Process 3 allocated starting at index 45
First Fit: Process 4 allocated starting at index 53
Memory Allocation: [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [1] [1] [1] [1]
[1] [1] [1] [1] [1] [1] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2]
 [2] [2] [3] [3] [3] [3] [3] [3] [3] [3] [4] [4] [4] [4] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [
] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [
 ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]

--- Best Fit ---
Best Fit: Process 0 allocated starting at index 85
```

```
--- Best Fit ---
Best Fit: Process 0 allocated starting at index 85
Best Fit: Process 1 allocated starting at index 75
Best Fit: Process 2 allocated starting at index 55
Best Fit: Process 3 allocated starting at index 47
Best Fit: Process 4 allocated starting at index 43
Memory Allocation: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
 [4] [4] [4] [4] [3] [3] [3] [3] [3] [3] [3] [3] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2

--- Next Fit ---
Next Fit: Process 0 allocated starting at index 0
Next Fit: Process 1 allocated starting at index 15
Next Fit: Process 2 allocated starting at index 25
Next Fit: Process 3 allocated starting at index 45
Next Fit: Process 4 allocated starting at index 53
Memory Allocation: [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [1] [1] [1] [1]
[1] [1] [1] [1] [1] [1] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2]
 [2] [2] [3] [3] [3] [3] [3] [3] [3] [3] [4] [4] [4] [4] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [
] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [
 ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
```

```
--- Worst Fit ---
Worst Fit: Process 0 allocated starting at index 0
Worst Fit: Process 1 allocated starting at index 15
Worst Fit: Process 2 allocated starting at index 25
Worst Fit: Process 3 allocated starting at index 45
Worst Fit: Process 4 allocated starting at index 53
Memory Allocation: [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [0] [1] [1] [1] [1]
[1] [1] [1] [1] [1] [1] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] [2]
 [2] [2] [3] [3] [3] [3] [3] [3] [3] [3] [4] [4] [4] [4] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [
] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [
 ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
```

Q] Implement Buddy System.

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

#define MAX_BLOCKS 10 // Maximum number of blocks
#define MIN_BLOCK_SIZE 16 // Minimum block size

typedef struct Block {
   int size;
   int free;
   struct Block *next;
} Block;

Block *freeList = NULL;

// Function to initialize the memory
void initializeMemory(int totalSize) {
   if (totalSize < MIN_BLOCK_SIZE) {
      printf("Total size must be at least %d bytes.\n", MIN_BLOCK_SIZE);
      return;
   }

   freeList = (Block *)malloc(sizeof(Block));
   freeList->size = totalSize;
   freeList->free = 1; // Mark as free
   freeList->next = NULL;
}

// Function to split a block into two buddies
void splitBlock(Block *block) {
   if (block->size <= MIN_BLOCK_SIZE) {
      return;
   }

   Block *buddy = (Block *)((char *)block + block->size / 2);
   buddy->size = block->size / 2;
   buddy->free = 1;
   buddy->next = block->next;

   block->size = block->size / 2;
   block->next = buddy;
}

// Function to merge two buddies
void mergeBlocks(Block *block) {
   Block *nextBlock = (Block *)((char *)block + block->size);
   if (nextBlock && nextBlock->free) {
      block->size += nextBlock->size;
      block->next = nextBlock->next; // Remove nextBlock from free list
      free(nextBlock);
   }
}
```

```
// Function to allocate memory
void *allocateMemory(int size) {
    Block *current = freeList;
    while (current) {
        if (current->free && current->size >= size) {
            while (current->size > size) {
                splitBlock(current);
            }
            current->free = 0; // Mark as allocated
            return (void *)((char *)current + sizeof(Block)); // Return pointer to usable memory
        }
        current = current->next;
    }
    return NULL; // No suitable block found
}

// Function to free memory
void freeMemory(void *ptr) {
    Block *block = (Block *)((char *)ptr - sizeof(Block));
    block->free = 1; // Mark as free
    mergeBlocks(block); // Merge with buddy if possible
}

// Function to display the free list
void displayFreeList() {
    Block *current = freeList;
    printf("Free List:\n");
    while (current) {
        printf("Block size: %d, Status: %s\n", current->size, current->free ? "Free" : "Allocated");
        current = current->next;
    }
}

int main() {
    int totalSize;

    printf("Enter total memory size (must be >= %d): ", MIN_BLOCK_SIZE);
    scanf("%d", &totalSize);

    initializeMemory(totalSize);

    // Sample allocations
    void *ptr1 = allocateMemory(20);
    printf("Allocated 20 bytes at %p\n", ptr1);
    displayFreeList();

    void *ptr2 = allocateMemory(30);
    printf("Allocated 30 bytes at %p\n", ptr2);
    displayFreeList();

    freeMemory(ptr1);
    printf("Freed 20 bytes at %p\n", ptr1);
    displayFreeList();
```

```
    void *ptr3 = allocateMemory(10);
    printf("Allocated 10 bytes at %p\n", ptr3);
    displayFreeList();

    freeMemory(ptr2);
    printf("Freed 30 bytes at %p\n", ptr2);
    displayFreeList();

    freeMemory(ptr3);
    printf("Freed 10 bytes at %p\n", ptr3);
    displayFreeList();

    // Cleanup
    free(freeList);

    return 0;
}
```

Output ===

```
Enter total memory size (must be >= 16): 256
Allocated 20 bytes at 00921B44
Free List:
Block size: 16, Status: Allocated
Block size: 16, Status: Free
Block size: 32, Status: Free
Block size: 64, Status: Free
Block size: 128, Status: Free
Allocated 30 bytes at 00921B64
Free List:
Block size: 16, Status: Allocated
Block size: 16, Status: Free
Block size: 16, Status: Allocated
Block size: 16, Status: Free
Block size: 64, Status: Free
Block size: 128, Status: Free
Freed 20 bytes at 00921B44
Free List:
Block size: 32, Status: Free
Block size: 16, Status: Allocated
Block size: 16, Status: Free
Block size: 64, Status: Free
Block size: 128, Status: Free
```

Q1] Page Replacement Algorithm First In First Out (FIFO), Least Recently Used (LRU), Optimal

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

#define MAX_PAGES 100
#define MAX_FRAMES 10

// Function to implement FIFO Page Replacement
void fifo(int pages[], int numPages, int numFrames) {
   int frame[MAX_FRAMES], pageFaults = 0, index = 0;
   for (int i = 0; i < numFrames; i++) frame[i] = -1; // Initialize frames

   for (int i = 0; i < numPages; i++) {
      int found = 0;
      for (int j = 0; j < numFrames; j++) {
         if (frame[j] == pages[i]) {
            found = 1; // Page is already in frame
            break;
         }
      }

      if (!found) {
         frame[index] = pages[i]; // Replace the page
         index = (index + 1) % numFrames; // Move to the next frame
         pageFaults++;
      }

      printf("FIFO Frame: ");
      for (int j = 0; j < numFrames; j++) {
         if (frame[j] != -1)
            printf("%d ", frame[j]);
      }
      printf("\n");
   }

   printf("Total Page Faults (FIFO): %d\n", pageFaults);
}

// Function to implement LRU Page Replacement
void lru(int pages[], int numPages, int numFrames) {
   int frame[MAX_FRAMES], pageFaults = 0, lastUsed[MAX_FRAMES], time = 0;
   for (int i = 0; i < numFrames; i++) frame[i] = -1; // Initialize frames

   for (int i = 0; i < numPages; i++) {
      int found = 0;
      for (int j = 0; j < numFrames; j++) {
         if (frame[j] == pages[i]) {
            found = 1; // Page is already in frame
            lastUsed[j] = time++; // Update last used time
            break;
         }
      }
```

```
        if (!found) {
            int lruIndex = 0;
            for (int j = 0; j < numFrames; j++) {
                if (frame[j] == -1) {
                    lruIndex = j; // Find an empty frame
                    break;
                }
                if (lastUsed[j] < lastUsed[lruIndex]) {
                    lruIndex = j; // Find the least recently used frame
                }
            }

            frame[lruIndex] = pages[i]; // Replace the page
            lastUsed[lruIndex] = time++;
            pageFaults++;
        }

        printf("LRU Frame: ");
        for (int j = 0; j < numFrames; j++) {
            if (frame[j] != -1)
                printf("%d ", frame[j]);
        }
        printf("\n");
    }

    printf("Total Page Faults (LRU): %d\n", pageFaults);
}

// Function to implement Optimal Page Replacement
void optimal(int pages[], int numPages, int numFrames) {
    int frame[MAX_FRAMES], pageFaults = 0;
    for (int i = 0; i < numFrames; i++) frame[i] = -1; // Initialize frames

    for (int i = 0; i < numPages; i++) {
        int found = 0;
        for (int j = 0; j < numFrames; j++) {
            if (frame[j] == pages[i]) {
                found = 1; // Page is already in frame
                break;
            }
        }

        if (!found) {
            int replaceIndex = -1, farthest = -1;
            for (int j = 0; j < numFrames; j++) {
                int k;
                for (k = i + 1; k < numPages; k++) {
                    if (frame[j] == pages[k]) {
                        if (k > farthest) {
                            farthest = k;
                            replaceIndex = j; // Found a frame to replace
                        }
                        break;
                    }
```

```
            }
            if (k == numPages) {
                replaceIndex = j; // Frame not used again
                break;
            }
        }

        frame[replaceIndex] = pages[i]; // Replace the page
        pageFaults++;
    }

    printf("Optimal Frame: ");
    for (int j = 0; j < numFrames; j++) {
        if (frame[j] != -1)
            printf("%d ", frame[j]);
    }
    printf("\n");
  }

  printf("Total Page Faults (Optimal): %d\n", pageFaults);
}

int main() {
  int pages[MAX_PAGES], numPages, numFrames;

  printf("Enter number of pages: ");
  scanf("%d", &numPages);
  printf("Enter page reference string:\n");
  for (int i = 0; i < numPages; i++) {
    scanf("%d", &pages[i]);
  }

  printf("Enter number of frames: ");
  scanf("%d", &numFrames);

  printf("\n--- FIFO Page Replacement ---\n");
  fifo(pages, numPages, numFrames);

  printf("\n--- LRU Page Replacement ---\n");
  lru(pages, numPages, numFrames);

  printf("\n--- Optimal Page Replacement ---\n");
  optimal(pages, numPages, numFrames);

  return 0;
}
```

Output ===

```
LRU_optimal_page_replacement }
Enter number of pages: 5
Enter page reference string:
7
2
3
6
3
Enter number of frames: 4

--- FIFO Page Replacement ---
FIFO Frame: 7
FIFO Frame: 7 2
FIFO Frame: 7 2 3
FIFO Frame: 7 2 3 6
FIFO Frame: 7 2 3 6
Total Page Faults (FIFO): 4

--- LRU Page Replacement ---
LRU Frame: 7
LRU Frame: 7 2
LRU Frame: 7 2 3
LRU Frame: 7 2 3 6
```

```
3
Enter number of frames: 4

--- FIFO Page Replacement ---
FIFO Frame: 7
FIFO Frame: 7 2
FIFO Frame: 7 2 3
FIFO Frame: 7 2 3 6
FIFO Frame: 7 2 3 6
Total Page Faults (FIFO): 4

--- LRU Page Replacement ---
LRU Frame: 7
LRU Frame: 7 2
LRU Frame: 7 2 3
LRU Frame: 7 2 3 6
LRU Frame: 7 2 3 6
Total Page Faults (LRU): 4

--- Optimal Page Replacement ---
Optimal Frame: 7
Optimal Frame: 2
Optimal Frame: 3
Optimal Frame: 3 6
Optimal Frame: 3 6
Total Page Faults (Optimal): 4
```