HAPPY BIRTHDAY KSHITIJ

```
1)#include <stdio.h>
#include <stdlib.h>
// Define the memory allocation block structure
struct mab {
    int offset;
    int size;
    int allocated;
    struct mab* next;
    struct mab* prev;
};
typedef struct mab Mab;
typedef Mab* MabPtr;
MabPtr memSplit(MabPtr m, int size);
MabPtr memMerge(MabPtr m);
// Function to check if memory is available
MabPtr memChk(MabPtr m, int size) {
    MabPtr current = m;
    while (current != NULL) {
        if (!current->allocated && current->size >= size) {
            return current;
        }
        current = current->next;
    }
    return NULL;
}
// Function to allocate memory using First Fit
MabPtr memAllocFirstFit(MabPtr m, int size) {
    MabPtr block = memChk(m, size);
    if (block != NULL) {
        if (block->size == size) {
            block->allocated = 1;
        } else {
            MabPtr newBlock = memSplit(block, size);
```

```
            newBlock->allocated = 1;
        }
    }
    return m;
}
// Function to allocate memory using Best Fit
MabPtr memAllocBestFit(MabPtr m, int size) {
    MabPtr bestBlock = NULL;
    MabPtr current = m;
    while (current != NULL) {
        if (!current->allocated && current->size >= size) {
            if (bestBlock == NULL || current->size < bestBlock->size) {
                bestBlock = current;
            }
        }
        current = current->next;
    }
    if (bestBlock != NULL) {
        if (bestBlock->size == size) {
            bestBlock->allocated = 1;
        } else {
            MabPtr newBlock = memSplit(bestBlock, size);
            newBlock->allocated = 1;
        }
    }
    return m;
}
// Function to allocate memory using Worst Fit
MabPtr memAllocWorstFit(MabPtr m, int size) {
    MabPtr worstBlock = NULL;
    MabPtr current = m;
    while (current != NULL) {
        if (!current->allocated && current->size >= size) {
            if (worstBlock == NULL || current->size > worstBlock->size) {
                worstBlock = current;
            }
        }
        current = current->next;
    }
```

```
    if (worstBlock != NULL) {
        if (worstBlock->size == size) {
            worstBlock->allocated = 1;
        } else {
            MabPtr newBlock = memSplit(worstBlock, size);
            newBlock->allocated = 1;
        }
    }
    return m;
}
// Function to free memory block
MabPtr memFree(MabPtr m) {
    m->allocated = 0;
    return memMerge(m);
}
// Function to merge two memory blocks
MabPtr memMerge(MabPtr m) {
    if (m->next != NULL && !m->next->allocated) {
        m->size += m->next->size;
        m->next = m->next->next;
        if (m->next != NULL) {
            m->next->prev = m;
        }
    }
    if (m->prev != NULL && !m->prev->allocated) {
        m->prev->size += m->size;
        m->prev->next = m->next;
        if (m->next != NULL) {
            m->next->prev = m->prev;
        }
        return m->prev;
    }
    return m;
}
// Function to split a memory block
MabPtr memSplit(MabPtr m, int size) {
    MabPtr newBlock = (MabPtr)malloc(sizeof(Mab));
    newBlock->offset = m->offset + size;
    newBlock->size = m->size - size;
```

```c
    newBlock->allocated = 0;
    newBlock->prev = m;
    newBlock->next = m->next;
    if (m->next != NULL) {
        m->next->prev = newBlock;
    }
    m->size = size;
    m->next = newBlock;
    return newBlock;
}
int main() {
    // Initialize memory blocks using your linked list structure
    MabPtr memory = (MabPtr)malloc(sizeof(Mab));
    memory->offset = 0;
    memory->size = 128;
    memory->allocated = 0;
    memory->next = NULL;
    memory->prev = NULL;
    // Example usage of memory allocation policies
    // First Fit
    printf("First Fit:\n");
    memory = memAllocFirstFit(memory, 64);
    memory = memAllocFirstFit(memory, 32);
    memory = memAllocFirstFit(memory, 16);
    MabPtr current = memory;
    while (current != NULL) {
        printf("Block: offset=%d, size=%d, allocated=%d\n", current->offset, current->size,
current->allocated);
        current = current->next;
    }
    // Reset memory for the next policy
    free(memory);
    memory = (MabPtr)malloc(sizeof(Mab));
    memory->offset = 0;
    memory->size = 128;
    memory->allocated = 0;
    memory->next = NULL;
    memory->prev = NULL;
```

```
    // Best Fit
    printf("\nBest Fit:\n");
    memory = memAllocBestFit(memory, 16);
    memory = memAllocBestFit(memory, 32);
    memory = memAllocBestFit(memory, 64);
    current = memory;
    while (current != NULL) {
        printf("Block: offset=%d, size=%d, allocated=%d\n", current->offset, current->size,
current->allocated);
        current = current->next;
    }
    // Reset memory for the next policy
    free(memory);
    memory = (MabPtr)malloc(sizeof(Mab));
    memory->offset = 0;
    memory->size = 128;
    memory->allocated = 0;
    memory->next = NULL;
    memory->prev = NULL;
    // Worst Fit
    printf("\nWorst Fit:\n");
    memory = memAllocWorstFit(memory, 64);
    memory = memAllocWorstFit(memory, 32);
    memory = memAllocWorstFit(memory, 16);
    current = memory;
    while (current != NULL) {
        printf("Block: offset=%d, size=%d, allocated=%d\n", current->offset, current->size,
current->allocated);
        current = current->next;
    }
    // Clean up allocated memory
    while (memory != NULL) {
        MabPtr temp = memory;
        memory = memory->next;
        free(temp);
    }
    return 0;
}
```

2)

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

// Structure to represent a process
struct Process {
    int id;
    int arrival_time;
    int burst_time;
    int current_queue;  // The current queue the process is in
};

typedef struct Process Process;

// Function to initialize a process
Process* createProcess(int id, int arrival_time, int burst_time, int current_queue) {
    Process* process = (Process*)malloc(sizeof(Process));
    process->id = id;
    process->arrival_time = arrival_time;
    process->burst_time = burst_time;
    process->current_queue = current_queue;
    return process;
}

// Function to perform Multilevel Feedback Queue scheduling for a queue
// Function to perform Multilevel Feedback Queue scheduling for a queue
void multilevelFeedbackQueue(Process** queue, int num_processes, int num_queues, int
time_quantum[]) {
    int time = 0;
    int completed_processes = 0;
    int current_queue = 0;

    while (completed_processes < num_processes) {
        int remaining = 0;
        for (int i = 0; i < num_processes; i++) {
            Process* process = queue[i];

            if (process->burst_time > 0) {
                remaining = 1;
```

```
            int quantum = time_quantum[process->current_queue];
            if (process->burst_time > quantum) {
                time += quantum;
                process->burst_time -= quantum;
            } else {
                time += process->burst_time;
                printf("Process %d completed (Queue %d) at time %d\n", process->id, process-
>current_queue, time);
                completed_processes++;
                process->burst_time = 0;
            }
        }

        // Demote the process to a lower priority queue
        if (process->burst_time > 0 && process->current_queue < num_queues - 1) {
            process->current_queue++;
        }
    }

    if (remaining == 0) {
        // No process remaining in the current queue, move to the next queue
        printf("Queue %d is empty at time %d\n", current_queue, time);
        current_queue = (current_queue + 1) % num_queues;
    }
  }
}


int main() {
  int num_processes = 5;
  int num_queues = 3;
  Process* processes[5];

  // Create sample processes
  processes[0] = createProcess(1, 0, 20, 0);
  processes[1] = createProcess(2, 2, 8, 0);
  processes[2] = createProcess(3, 4, 76, 0);
  processes[3] = createProcess(4, 6, 7, 0);
  processes[4] = createProcess(5, 8, 5, 0);
```

```
    // Define time quantum for each queue
    int time_quantum[] = {4, 8, 16};

    printf("Processes scheduled using Multilevel Feedback Queue:\n");
    multilevelFeedbackQueue(processes, num_processes, num_queues, time_quantum);

    // Free allocated memory
    for (int i = 0; i < num_processes; i++) {
        free(processes[i]);
    }

    return 0;
}
```

3)

```
#include <stdio.h>
// Define the maximum segment size
#define MAX_SEGMENT_SIZE 1024
// Define the number of segments
#define NUM_SEGMENTS 5
// Structure to represent a segment entry in the segment table
typedef struct {
    int base;  // Base address of the segment in physical memory
    int limit; // Size of the segment
} SegmentEntry;
// Function to create a segment table with random base addresses and limits
void createSegmentTable(SegmentEntry segmentTable[], int numSegments) {
    // Randomly assign base addresses and limits
    for (int i = 0; i < numSegments; i++) {
        segmentTable[i].base = i * MAX_SEGMENT_SIZE;
        segmentTable[i].limit = MAX_SEGMENT_SIZE;
    }
}
// Function to convert logical address to physical address
int convertToPhysicalAddress(SegmentEntry segmentTable[], int segment, int offset) {
    if (segment < 0 || segment >= NUM_SEGMENTS) {
        printf("Segment number out of range\n");
```

```
        return -1;
    }
    if (offset < 0 || offset >= segmentTable[segment].limit) {
        printf("Offset is out of the segment's range\n");
        return -1;
    }
    int physicalAddress = segmentTable[segment].base + offset;
    return physicalAddress;
}
int main() {
    // Create a segment table
    SegmentEntry segmentTable[NUM_SEGMENTS];
    createSegmentTable(segmentTable, NUM_SEGMENTS);
    // Compute physical addresses for the given scenarios
    int segment, offset, physicalAddress;
    // Scenario (i): 53 bytes of segment 2
    segment = 2;
    offset = 53;
    physicalAddress = convertToPhysicalAddress(segmentTable, segment, offset);
    if (physicalAddress != -1) {
        printf("(i) Physical Address: %d\n", physicalAddress);
    }
    // Scenario (ii): 852 bytes of segment 3
    segment = 3;
    offset = 852;
    physicalAddress = convertToPhysicalAddress(segmentTable, segment, offset);
    if (physicalAddress != -1) {
        printf("(ii) Physical Address: %d\n", physicalAddress);
    }
    // Scenario (iii): 1222 bytes of segment 0
    segment = 0;
    offset = 1222;
    if (offset >= 0 && offset < segmentTable[segment].limit) {
        physicalAddress = convertToPhysicalAddress(segmentTable, segment, offset);
        printf("(iii) Physical Address: %d\n", physicalAddress);
    }
    return 0;
}
```

4)

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_FRAMES 4
#define MAX_PAGES 12
// Structure to represent a page frame
typedef struct {
    int page_number;
    int second_chance_bit;
} PageFrame;
// Function to initialize the page frames
void initializeFrames(PageFrame frames[], int num_frames) {
    for (int i = 0; i < num_frames; i++) {
        frames[i].page_number = -1;
        frames[i].second_chance_bit = 0;
    }
}
// Function to check if a page is present in the frames
int isPageInFrames(PageFrame frames[], int num_frames, int page_number) {
    for (int i = 0; i < num_frames; i++) {
        if (frames[i].page_number == page_number) {
            return 1;
        }
    }
    return 0;
}
// Function to find the index of the page to be replaced (using the second chance algorithm)
int findReplacementIndex(PageFrame frames[], int num_frames, int current_index) {
    int index = current_index;
    while (1) {
        if (frames[index].second_chance_bit == 0) {
            return index;
        } else {
            frames[index].second_chance_bit = 0; // Give it a second chance
            index = (index + 1) % num_frames;
        }
    }
}
```

```
int main() {
    int page_references[MAX_PAGES] = {1, 2, 3, 2, 4, 3, 5, 6, 7, 6, 8, 1};
    PageFrame frames[MAX_FRAMES];
    int num_page_faults = 0;
    int current_frame_index = 0;
    initializeFrames(frames, MAX_FRAMES);
    printf("Page References: ");
    for (int i = 0; i < MAX_PAGES; i++) {
        int page_number = page_references[i];
        printf("%d ", page_number);
        if (!isPageInFrames(frames, MAX_FRAMES, page_number)) {
            num_page_faults++;
            int replacement_index = findReplacementIndex(frames, MAX_FRAMES,
current_frame_index);
            frames[replacement_index].page_number = page_number;
            frames[replacement_index].second_chance_bit = 0;
            current_frame_index = (replacement_index + 1) % MAX_FRAMES;
        } else {
            for (int j = 0; j < MAX_FRAMES; j++) {
                if (frames[j].page_number == page_number) {
                    frames[j].second_chance_bit = 1;
                    break;
                }
            }
        }
    }
    float hit_ratio = 1.0 - (float)num_page_faults / MAX_PAGES;
    printf("\nTotal Page Faults: %d\n", num_page_faults);
    printf("Hit Ratio: %.2f\n", hit_ratio);
    return 0;
}
```