Student: Hunter Antal
ID: 1181729
Class: 3655

Lab 4

Part IV: Description of Tasks

Task 1 (Concept Questions, Total Marks: 20)

A:

- Standard contiguous preallocates a block of memory to the files, this makes retrieving the files fast but if the files don't use all the space then you have holes in your memory.
- Linked implementations assign noncontiguous blocks linking them with pointers. This stops holes but makes retrieval slower because of pointer traversal.
- The overflow method preallocates a block of memory then if the files need more space allocate noncontiguous memory using pointers. This is the best of both worlds because it's still fast to access and you can always have more space if needed.

B:

(i) Mounting a file system at multiple locations can lead to consistency issues if files are modified in one location but not in others, data corruption from simultaneous conflicting updates, security risks as access control may be compromised, and file locking issues where concurrent accesses cause conflicts in file usage.

(ii) Using a shared stack for parameter passing can lead to data corruption if concurrent processes overwrite stack data, security risks where unauthorized processes access sensitive information, and unintended data leakage if one process's data remains accessible to another.

Task 2

A:

```
// Created by: Hunter Antal
// Created on: 15/11/2024
// Student ID: 1181729
#include <stdio.h>
#include <stdlib.h>


#define MAX 10
```

```
void loadPages(int program[], int size, int storage[], int memory[]);

int main()
{
    // this is a simple program that simulates the memory management of a computer system
    // this is an idealized version of the memory management system of a computer where
there are enough frames for each page

    // Secoundary storage each element represents a page in the main storage
    int storage[MAX] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    // program 1 (word) uses 3 pages
    int program1[3] = {1, 2, 3};

    // program 2 (excel) uses 4 pages
    int program2[4] = {4, 5, 6, 7};

    // program 3 (power point) uses 3 pages
    int program3[3] = {8, 9, 10};

    // Main memory containing 10 frames
    int memory[MAX] = {0};

    // holds users choice of program
    int program = -1;
    while (program != 0)
    {
        // ask user which program to load
        printf("Programs available:\n");
        printf("0. Close\n");
        printf("1. Word\n");
        printf("2. Excel\n");
        printf("3. Power Point\n");
        printf("Enter program to load: ");
        scanf("%d", &program);

        // switch case to load program
        switch (program)
        {
        case 1:
            loadPages(program1, sizeof(program1) / sizeof(program1[0]), storage, memory);
```

```
            break;
        case 2:
            loadPages(program2, sizeof(program2) / sizeof(program2[0]), storage, memory);
            break;
        case 3:
            loadPages(program3, sizeof(program3) / sizeof(program3[0]), storage, memory);
            break;
        case 0:
            printf("\n");

printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n")
;
            printf("Closing program\n");

printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n")
;
            printf("\n");
            break;
        default:
            printf("\n");

printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n")
;
            printf("Invalid Input: %d\n", program);

printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n")
;
            printf("\n");
            break;
        }
    }

    return 0;
}

// function to load a program into memory
void loadPages(int program[], int size, int storage[], int memory[])
{

    if (size > MAX)
    {
        printf("Program size is greater than main memory size\n");
```

```
        return;
    }
    else
    {
        // Load pages from storage into main memory
        for (int i = 0; i < size; i++)
        {
            memory[program[i] - 1] = storage[program[i] - 1];
        }
    }

    // print main memory
    printf("\n");

printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n")
;
    printf("Main memory: ");
    printf("[");
    for (int i = 0; i < MAX; i++)
    {
        printf("%d", memory[i]);
        if (i != MAX - 1)
            printf(", ");
    }
    printf("]");
    printf("\n");

printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n")
;
    printf("\n");
}
```

B:

```
// Created by: Hunter Antal
// Created on: 16/11/2024
// Student ID: 1181729
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Define the structure for a linked list node
```

```
typedef struct node
{
    int val;           // Value stored in the node
    struct node *next; // Pointer to the next node
} node_t;

// Function to print the entire linked list
void print_list(node_t *head)
{
    printf("\n");
    printf("Printing List:\n");

printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n")
;

    node_t *current = head;

    while (current != NULL) // Traverse the list
    {
        printf("%d\n", current->val); // Print the value of each node
        current = current->next;      // Move to the next node
    }

printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n")
;
}

// Function to add a new node at the end of the list
void add(node_t *head, int val)
{
    if (head == NULL) // Check if head is NULL
    {
        perror("Head is NULL in add()");
        exit(EXIT_FAILURE);
    }

    node_t *current = head;
    // Traverse to the last node
    while (current->next != NULL)
    {
        current = current->next;
    }
```

```
    // Allocate memory for the new node
    current->next = (node_t *)malloc(sizeof(node_t));
    if (current->next == NULL) // Check for memory allocation failure
    {
        perror("Memory allocation failed");
        exit(EXIT_FAILURE);
    }

    // Initialize the new node
    current->next->val = val;
    current->next->next = NULL;
}

// Function to add a node at the beginning of the list
void push(node_t **head, int val)
{
    // Allocate memory for the new head node
    node_t *newHead = (node_t *)malloc(sizeof(node_t));
    if (newHead == NULL) // Check for memory allocation failure
    {
        perror("Memory allocation failed");
        exit(EXIT_FAILURE);
    }

    // Set the new head's value and point it to the current head
    newHead->next = *head;
    newHead->val = val;

    *head = newHead; // Update the head pointer
}

// Function to remove the head of the list (LRU implementation)
int pop(node_t **head)
{
    if (*head == NULL) // Check if the list is empty
    {
        printf("List is empty, nothing to pop.\n");
        return -1; // Return error value
    }

    int val = (*head)->val; // Store the value of the head
```

```
    node_t *temp = *head;   // Temporarily hold the head
    *head = (*head)->next;  // Move the head pointer to the next node
    free(temp);             // Free the old head node

    return val; // Return the value of the removed node
}

// Function to search for a value in the linked list
int search(node_t **head, int val)
{
    node_t *current = *head;
    node_t *prev = NULL;

    // Traverse the list to find the page
    while (current != NULL)
    {
        if (current->val == val) // Page found
        {
            // If the page is already at the head, do nothing
            if (prev == NULL)
                return 1;

            // Move the page to the front
            prev->next = current->next; // Remove the node from its current position
            current->next = *head;      // Insert it at the front
            *head = current;            // Update head

            return 1; // Hit
        }
        prev = current;
        current = current->next;
    }
    return 0; // Page fault
}

/*Example input
How many frames would you like to use?: 3
Enter page sequence separated by spaces (end with newline): 7 0 1 2 0 3 0 4 2 3
*/

int main()
{
```

```
    int hits = 0;          // Count of hits
    int pageFaults = 0;    // Count of page faults
    int frameCount = 0;    // Current number of frames in use
    int pageReference = 0; // Total number of page references
    float hitRatio = 0.0;  // Hit ratio
    while (1 == 1)
    {
        printf("\n");

printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n")
;
        printf("Welcome to the LRU page replacement algorithm simulation\n");

printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n")
;
        printf("\n");
        printf("If you would like to exit the program at any time, press Ctrl+C\n");
        printf("\n");
        // Prompt user for the number of frames
        printf("How many frames would you like to use?: ");
        int numberOfFrames = 0;
        scanf("%d", &numberOfFrames);
        while(numberOfFrames <= 0)
        {
            printf("Invalid number of frames\n");
            printf("How many frames would you like to use?: ");
            scanf("%d", &numberOfFrames);
        }

        // Prompt user for the page sequence
        printf("Enter page sequence separated by spaces (end with newline): ");
        char input[1000]; // Buffer for user input
        int *arr = NULL;  // Dynamic array to store the page sequence
        int n = 0;        // Number of elements in the page sequence

        getchar();                              // Clear newline left by scanf
        if (fgets(input, sizeof(input), stdin) == NULL) // Read input as a line
        {
            perror("Failed to read input");
            exit(EXIT_FAILURE);
        }
```

```
// Tokenize the input and convert to integers
char *token = strtok(input, " ");
while (token != NULL)
{
    arr = realloc(arr, (n + 1) * sizeof(int)); // Dynamically resize the array
    if (arr == NULL)                           // Check for memory allocation failure
    {
        perror("Memory allocation failed");
        exit(EXIT_FAILURE);
    }

    arr[n++] = atoi(token); // Convert token to integer
    token = strtok(NULL, " ");
}

// Initialize the linked list
node_t *head = (node_t *)malloc(sizeof(node_t));
if (head == NULL) // Check for memory allocation failure
{
    perror("Memory allocation failed");
    exit(EXIT_FAILURE);
}

head->val = -1; // Dummy value for the head
head->next = NULL;

// Simulate the page replacement algorithm
for (int i = 0; i < n; i++)
{
    pageReference++;          // Increment page reference count
    if (search(&head, arr[i])) // Check if the page is already in memory
    {
        hits++; // Increment hits if found
    }
    else
    {
        pageFaults++;                  // Increment page faults if not found
        if (frameCount < numberOfFrames) // If frames are available
        {
            add(head, arr[i]); // Add the page to memory
            frameCount++;      // Increment the frame count
        }
```

```
            else // If no frames are available, remove LRU and add new page
            {
                pop(&head);
                add(head, arr[i]);
            }
        }
    }

    // Calculate and display the hit ratio
    hitRatio = (float)hits / pageReference;
    printf("Number of hits: %d\n", hits);
    printf("Number of page faults: %d\n", pageFaults);
    printf("Hit ratio: %.2f\n", hitRatio);

    free(arr);  // Free the dynamic array
    free(head); // Free the linked list
  }
  return 0;
}
```