Slip no-1

Q.1 Write the simulation program to implement demand paging and show the page scheduling and total number of page faults according to the LFU page replacement algorithm. Assume the memory of n frames.
Reference String : 3,4,5,4,3,4,7,2,4,5,6,7,2,4,6

```c
#include <stdio.h>

#define MAX 100

int findLFU(int freq[], int n, int frames[], int size) {
    int min = freq[frames[0]], minIndex = 0;

    for (int i = 1; i < size; i++) {
        if (freq[frames[i]] < min) {
            min = freq[frames[i]];
            minIndex = i;
        }
    }
    return minIndex;
}

int main() {
    int n, frames[MAX], freq[MAX] = {0}, pages[MAX], page_faults = 0, size
= 0;

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

    printf("Enter the number of pages: ");
    int page_count;
    scanf("%d", &page_count);

    printf("Enter the reference string: ");
    for (int i = 0; i < page_count; i++) {
        scanf("%d", &pages[i]);
    }
```

```
    for (int i = 0; i < page_count; i++) {
        int page = pages[i];
        freq[page]++;

        int found = 0;
        for (int j = 0; j < size; j++) {
            if (frames[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            if (size < n) {
                frames[size++] = page;
            } else {
                int lfuIndex = findLFU(freq, n, frames, size);
                frames[lfuIndex] = page;
            }
            page_faults++;
        }

        printf("Frames: ");
        for (int j = 0; j < size; j++) {
            printf("%d ", frames[j]);
        }
        printf("\n");
    }

    printf("Total page faults: %d\n", page_faults);

    return 0;
}
```

Q.2 Write a C program to implement the shell which displays the command prompt “myshell$”. It accepts the command, tokenize the command line and execute it by creating the child process. Also implement the additional command ‘typeline’ as

typeline +n filename :- To print first n lines in the file.
typeline -a filename :- To print all lines in the file.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <fcntl.h>

#define MAX 100

void typeline(char *option, char *filename) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return;
    }

    char line[MAX];
    if (strcmp(option, "-a") == 0) {
        while (fgets(line, MAX, file) != NULL) {
            printf("%s", line);
        }
    } else if (option[0] == '+' && strlen(option) > 1) {
        int n = atoi(option + 1);
        for (int i = 0; i < n && fgets(line, MAX, file) != NULL; i++) {
            printf("%s", line);
        }
    }

    fclose(file);
}

int main() {
    char input[MAX];
    char *args[MAX];
```

```
    while (1) {
        printf("myshell$ ");
        fgets(input, MAX, stdin);
        input[strlen(input) - 1] = '\0'; // remove newline character

        int i = 0;
        args[i] = strtok(input, " ");
        while (args[i] != NULL) {
            i++;
            args[i] = strtok(NULL, " ");
        }

        if (strcmp(args[0], "exit") == 0) {
            break;
        } else if (strcmp(args[0], "typeline") == 0 && args[1] && args[2])
{
            typeline(args[1], args[2]);
        } else {
            pid_t pid = fork();
            if (pid == 0) {
                execvp(args[0], args);
                perror("Error executing command");
                exit(1);
            } else {
                wait(NULL);
            }
        }
    }

    return 0;
}
```

Slip no-2

Q.1 Write the simulation program for demand paging and show the page scheduling and total number of page faults according the FIFO page replacement algorithm. Assume the memory of n frames.
Reference String : 3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6

```c
#include <stdio.h>

#define MAX 100

int main() {
    int n, frames[MAX], pages[MAX], page_faults = 0, index = 0;
    int page_count;

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

    printf("Enter the number of pages in the reference string: ");
    scanf("%d", &page_count);

    printf("Enter the reference string: ");
    for (int i = 0; i < page_count; i++) {
        scanf("%d", &pages[i]);
    }

    for (int i = 0; i < n; i++) {
        frames[i] = -1;
    }

    for (int i = 0; i < page_count; i++) {
        int page = pages[i];
        int found = 0;

        for (int j = 0; j < n; j++) {
            if (frames[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            frames[index] = page;
            index = (index + 1) % n;
            page_faults++;
```

```
            printf("Page %d loaded into frame. Current frames: ", page);
            for (int j = 0; j < n; j++) {
                if (frames[j] != -1) {
                    printf("%d ", frames[j]);
                } else {
                    printf("- ");
                }
            }
            printf("\n");
        }
    }

    printf("Total page faults: %d\n", page_faults);

    return 0;
}
```

Q.2 Write a program to implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following ‘list’ commands as
myshell$ list f dirname :- To print names of all the files in current directory.
myshell$ list n dirname :- To print the number of all entries in the current directory

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <dirent.h>

#define MAX 100

void list_files(char *dirname) {
    struct dirent *entry;
```

```
    DIR *dir = opendir(dirname);

    if (dir == NULL) {
        perror("Error opening directory");
        return;
    }

    while ((entry = readdir(dir)) != NULL) {
        if (entry->d_type == DT_REG) { // Only list regular files
            printf("%s\n", entry->d_name);
        }
    }

    closedir(dir);
}

void count_entries(char *dirname) {
    struct dirent *entry;
    DIR *dir = opendir(dirname);
    int count = 0;

    if (dir == NULL) {
        perror("Error opening directory");
        return;
    }

    while ((entry = readdir(dir)) != NULL) {
        count++;
    }

    printf("Total entries: %d\n", count);
    closedir(dir);
}

int main() {
    char input[MAX];
    char *args[MAX];

    while (1) {
        printf("myshell$ ");
```

```
        fgets(input, MAX, stdin);
        input[strlen(input) - 1] = '\0'; // Remove newline character

        int i = 0;
        args[i] = strtok(input, " ");
        while (args[i] != NULL) {
            i++;
            args[i] = strtok(NULL, " ");
        }

        if (args[0] == NULL) {
            continue;
        }

        if (strcmp(args[0], "exit") == 0) {
            break;
        } else if (strcmp(args[0], "list") == 0 && args[1] != NULL &&
args[2] != NULL) {
            if (strcmp(args[1], "f") == 0) {
                list_files(args[2]);
            } else if (strcmp(args[1], "n") == 0) {
                count_entries(args[2]);
            } else {
                printf("Invalid list command option.\n");
            }
        } else {
            pid_t pid = fork();
            if (pid == 0) {
                execvp(args[0], args);
                perror("Error executing command");
                exit(1);
            } else {
                wait(NULL);
            }
        }
    }

    return 0;
}
```

Slip no-3

Q.1 Write the simulation program to implement demand paging and show the page scheduling and total number of page faults according to the LRU (using counter method) page replacement algorithm. Assume the memory of n frames.
Reference String : 3,5,7,2,5,1,2,3,1,3,5,3,1,6,2

```c
#include <stdio.h>

#define MAX 100

int findLRU(int time[], int n) {
    int min = time[0], minIndex = 0;
    for (int i = 1; i < n; i++) {
        if (time[i] < min) {
            min = time[i];
            minIndex = i;
        }
    }
    return minIndex;
}

int main() {
    int frames[MAX], time[MAX], pages[MAX], page_faults = 0, n,
page_count, counter = 0;

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

    printf("Enter the number of pages in the reference string: ");
    scanf("%d", &page_count);

    printf("Enter the reference string: ");
    for (int i = 0; i < page_count; i++) {
        scanf("%d", &pages[i]);
    }
```

```
    for (int i = 0; i < n; i++) {
        frames[i] = -1; // Initialize frames as empty
    }

    for (int i = 0; i < page_count; i++) {
        int page = pages[i], found = 0;

        // Check if page is already in frames
        for (int j = 0; j < n; j++) {
            if (frames[j] == page) {
                found = 1;
                time[j] = ++counter; // Update time for LRU
                break;
            }
        }

        // If page not found, page fault occurs
        if (!found) {
            int index;
            if (i < n) {
                index = i; // Fill the frame initially
            } else {
                index = findLRU(time, n); // Find LRU page to replace
            }

            frames[index] = page;
            time[index] = ++counter;
            page_faults++;

            // Print current state of frames
            printf("Page %d loaded into frame. Current frames: ", page);
            for (int j = 0; j < n; j++) {
                if (frames[j] != -1) {
                    printf("%d ", frames[j]);
                } else {
                    printf("- ");
                }
            }
            printf("\n");
        }
```

```
    }

    printf("Total page faults: %d\n", page_faults);

    return 0;
}
```

Q.2 Write a programto implement the toy shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following commands.
count c filename :- To print number of characters in the file.
count w filename :- To print number of words in the file.
count l filename :- To print number of lines in the file.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <ctype.h>

#define MAX 100

void count_chars(char *filename) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return;
    }
    int char_count = 0;
    char ch;
    while ((ch = fgetc(file)) != EOF) {
        char_count++;
    }
    fclose(file);
    printf("Number of characters: %d\n", char_count);
}

void count_words(char *filename) {
```

```
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return;
    }
    int word_count = 0;
    char ch, prev = ' ';
    while ((ch = fgetc(file)) != EOF) {
        if (isspace(ch) && !isspace(prev)) {
            word_count++;
        }
        prev = ch;
    }
    if (!isspace(prev)) {
        word_count++;
    }
    fclose(file);
    printf("Number of words: %d\n", word_count);
}

void count_lines(char *filename) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return;
    }
    int line_count = 0;
    char ch;
    while ((ch = fgetc(file)) != EOF) {
        if (ch == '\n') {
            line_count++;
        }
    }
    fclose(file);
    printf("Number of lines: %d\n", line_count);
}

int main() {
    char input[MAX];
    char *args[MAX];
```

```
    while (1) {
        printf("myshell$ ");
        fgets(input, MAX, stdin);
        input[strlen(input) - 1] = '\0';

        int i = 0;
        args[i] = strtok(input, " ");
        while (args[i] != NULL) {
            i++;
            args[i] = strtok(NULL, " ");
        }

        if (args[0] == NULL) {
            continue;
        }

        if (strcmp(args[0], "exit") == 0) {
            break;
        } else if (strcmp(args[0], "count") == 0 && args[1] != NULL &&
args[2] != NULL) {
            if (strcmp(args[1], "c") == 0) {
                count_chars(args[2]);
            } else if (strcmp(args[1], "w") == 0) {
                count_words(args[2]);
            } else if (strcmp(args[1], "l") == 0) {
                count_lines(args[2]);
            } else {
                printf("Invalid count option.\n");
            }
        } else {
            pid_t pid = fork();
            if (pid == 0) {
                execvp(args[0], args);
                perror("Error executing command");
                exit(1);
            } else {
                wait(NULL);
            }
        }
```

```
    }

    return 0;
}
```

Slip no-4

Q.1 Write the simulation program for demand paging and show the page scheduling and total number of page faults according the MFU page replacement algorithm. Assume the memory of n frames.
Reference String : 8, 5, 7, 8, 5, 7, 2, 3, 7, 3, 5, 9, 4, 6, 2

```
#include <stdio.h>

#define MAX 100

int findMFU(int freq[], int n) {
    int max = freq[0], maxIndex = 0;
    for (int i = 1; i < n; i++) {
        if (freq[i] > max) {
            max = freq[i];
            maxIndex = i;
        }
    }
    return maxIndex;
}

int main() {
    int frames[MAX], pages[MAX], freq[MAX] = {0}, page_faults = 0, n,
page_count;

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

    printf("Enter the number of pages in the reference string: ");
    scanf("%d", &page_count);

    printf("Enter the reference string: ");
```

```
    for (int i = 0; i < page_count; i++) {
        scanf("%d", &pages[i]);
    }

    for (int i = 0; i < n; i++) {
        frames[i] = -1;
    }

    for (int i = 0; i < page_count; i++) {
        int page = pages[i], found = 0;

        for (int j = 0; j < n; j++) {
            if (frames[j] == page) {
                found = 1;
                freq[j]++;
                break;
            }
        }

        if (!found) {
            int index;
            if (i < n) {
                index = i;
            } else {
                index = findMFU(freq, n);
            }

            frames[index] = page;
            freq[index] = 1;
            page_faults++;

            printf("Page %d loaded into frame. Current frames: ", page);
            for (int j = 0; j < n; j++) {
                if (frames[j] != -1) {
                    printf("%d ", frames[j]);
                } else {
                    printf("- ");
                }
            }
            printf("\n");
```

```
        }
    }

    printf("Total page faults: %d\n", page_faults);

    return 0;
}
```

Q.2 Write a program to implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following commands.
myshell$ search a filename pattern :- To search all the occurrence of pattern in the file.
myshell$ search c filename pattern :- To count the number of occurrence of pattern in the file.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

#define MAX 100

void search_all(char *filename, char *pattern) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return;
    }

    char line[MAX];
    int line_number = 1;

    while (fgets(line, sizeof(line), file) != NULL) {
        if (strstr(line, pattern) != NULL) {
            printf("Pattern found at line %d: %s", line_number, line);
```

```
        }
        line_number++;
    }
    fclose(file);
}

void search_count(char *filename, char *pattern) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return;
    }

    char line[MAX];
    int count = 0;

    while (fgets(line, sizeof(line), file) != NULL) {
        char *ptr = line;
        while ((ptr = strstr(ptr, pattern)) != NULL) {
            count++;
            ptr += strlen(pattern);
        }
    }

    printf("Pattern '%s' found %d times\n", pattern, count);
    fclose(file);
}

int main() {
    char input[MAX];
    char *args[MAX];

    while (1) {
        printf("myshell$ ");
        fgets(input, MAX, stdin);
        input[strlen(input) - 1] = '\0';

        int i = 0;
        args[i] = strtok(input, " ");
        while (args[i] != NULL) {
```

```
            i++;
            args[i] = strtok(NULL, " ");
        }

        if (args[0] == NULL) {
            continue;
        }

        if (strcmp(args[0], "exit") == 0) {
            break;
        } else if (strcmp(args[0], "search") == 0 && args[1] != NULL &&
args[2] != NULL && args[3] != NULL) {
            if (strcmp(args[1], "a") == 0) {
                search_all(args[2], args[3]);
            } else if (strcmp(args[1], "c") == 0) {
                search_count(args[2], args[3]);
            } else {
                printf("Invalid search option.\n");
            }
        } else {
            pid_t pid = fork();
            if (pid == 0) {
                execvp(args[0], args);
                perror("Error executing command");
                exit(1);
            } else {
                wait(NULL);
            }
        }
    }

    return 0;
}
```

Slip no-5

Q.1 Write the simulation program for demand paging and show the page scheduling and total number of page faults according the optimal page replacement algorithm. Assume the memory of n frames.
Reference String : 8, 5, 7, 8, 5, 7, 2, 3, 7, 3, 5, 9, 4, 6, 2

```c
#include <stdio.h>

#define MAX 100

int findOptimal(int frames[], int ref[], int n, int currentIndex, int
ref_len) {
    int farthest = currentIndex, index = -1;
    for (int i = 0; i < n; i++) {
        int j;
        for (j = currentIndex + 1; j < ref_len; j++) {
            if (frames[i] == ref[j]) {
                if (j > farthest) {
                    farthest = j;
                    index = i;
                }
                break;
            }
        }
        if (j == ref_len) {
            return i;
        }
    }
    return (index == -1) ? 0 : index;
}

int main() {
    int frames[MAX], ref[MAX], page_faults = 0, n, ref_len;

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

    printf("Enter the number of pages in the reference string: ");
    scanf("%d", &ref_len);
```

```
    printf("Enter the reference string: ");
    for (int i = 0; i < ref_len; i++) {
        scanf("%d", &ref[i]);
    }

    for (int i = 0; i < n; i++) {
        frames[i] = -1;
    }

    for (int i = 0; i < ref_len; i++) {
        int page = ref[i], found = 0;

        for (int j = 0; j < n; j++) {
            if (frames[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            if (i < n) {
                frames[i] = page;
            } else {
                int index = findOptimal(frames, ref, n, i, ref_len);
                frames[index] = page;
            }
            page_faults++;

            printf("Page %d loaded. Current frames: ", page);
            for (int j = 0; j < n; j++) {
                if (frames[j] != -1) {
                    printf("%d ", frames[j]);
                } else {
                    printf("- ");
                }
            }
            printf("\n");
        }
    }
```

```
    printf("Total page faults: %d\n", page_faults);

    return 0;
}
```

Q.2 Write a program to implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following commands.
myshell$ search f filename pattern :- To display first occurrence of pattern in the file.
myshell$ search c filename pattern :- To count the number of occurrence of pattern in the file.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

#define MAX 100

void search_first(char *filename, char *pattern) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return;
    }

    char line[MAX];
    int line_number = 1;

    while (fgets(line, sizeof(line), file) != NULL) {
        if (strstr(line, pattern) != NULL) {
            printf("First occurrence of '%s' found at line %d: %s",
pattern, line_number, line);
            fclose(file);
            return;
```

```
        }
        line_number++;
    }

    printf("Pattern '%s' not found in the file.\n", pattern);
    fclose(file);
}

void search_count(char *filename, char *pattern) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return;
    }

    char line[MAX];
    int count = 0;

    while (fgets(line, sizeof(line), file) != NULL) {
        char *ptr = line;
        while ((ptr = strstr(ptr, pattern)) != NULL) {
            count++;
            ptr += strlen(pattern);
        }
    }

    printf("Pattern '%s' found %d times\n", pattern, count);
    fclose(file);
}

int main() {
    char input[MAX];
    char *args[MAX];

    while (1) {
        printf("myshell$ ");
        fgets(input, MAX, stdin);
        input[strlen(input) - 1] = '\0';  // Remove newline character

        int i = 0;
```

```
        args[i] = strtok(input, " ");
        while (args[i] != NULL) {
            i++;
            args[i] = strtok(NULL, " ");
        }

        if (args[0] == NULL) {
            continue;  // Empty command
        }

        if (strcmp(args[0], "exit") == 0) {
            break;  // Exit the shell
        } else if (strcmp(args[0], "search") == 0 && args[1] != NULL &&
args[2] != NULL && args[3] != NULL) {
            if (strcmp(args[1], "f") == 0) {
                search_first(args[2], args[3]);
            } else if (strcmp(args[1], "c") == 0) {
                search_count(args[2], args[3]);
            } else {
                printf("Invalid search option.\n");
            }
        } else {
            pid_t pid = fork();
            if (pid == 0) {
                execvp(args[0], args);
                perror("Error executing command");
                exit(1);
            } else {
                wait(NULL);
            }
        }
    }

    return 0;
}
```

Slip no-6

Q.1 Write the simulation program for demand paging and show the page scheduling and total number of page faults according the MRU page replacement algorithm. Assume the memory of n frames.
Reference String : 8, 5, 7, 8, 5, 7, 2, 3, 7, 3, 5, 9, 4, 6, 2

```c
#include <stdio.h>

#define MAX 100

int findMRU(int frames[], int ref[], int n, int currentIndex, int ref_len)
{
    int mruIndex = 0, maxTime = -1;

    for (int i = 0; i < n; i++) {
        int j;
        for (j = currentIndex - 1; j >= 0; j--) {
            if (frames[i] == ref[j]) {
                if (j > maxTime) {
                    maxTime = j;
                    mruIndex = i;
                }
                break;
            }
        }
    }
    return mruIndex;
}

int main() {
    int frames[MAX], ref[MAX], page_faults = 0, n, ref_len;

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

    printf("Enter the number of pages in the reference string: ");
    scanf("%d", &ref_len);

    printf("Enter the reference string: ");
    for (int i = 0; i < ref_len; i++) {
        scanf("%d", &ref[i]);
```

```
    }

    for (int i = 0; i < n; i++) {
        frames[i] = -1;
    }

    for (int i = 0; i < ref_len; i++) {
        int page = ref[i], found = 0;

        for (int j = 0; j < n; j++) {
            if (frames[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            if (i < n) {
                frames[i] = page;
            } else {
                int index = findMRU(frames, ref, n, i, ref_len);
                frames[index] = page;
            }
            page_faults++;

            printf("Page %d loaded. Current frames: ", page);
            for (int j = 0; j < n; j++) {
                if (frames[j] != -1) {
                    printf("%d ", frames[j]);
                } else {
                    printf("- ");
                }
            }
            printf("\n");
        }
    }

    printf("Total page faults: %d\n", page_faults);

    return 0;
```

```
}
```

Q.2 Write a programto implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following commands.
myshell$ search f filename pattern :- To display first occurrence of pattern in the file.
myshell$ search a filename pattern :- To search all the occurrence of pattern in the file.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

#define MAX 1024

void search_first(const char *filename, const char *pattern) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return;
    }

    char line[MAX];
    int line_number = 1;

    while (fgets(line, sizeof(line), file) != NULL) {
        if (strstr(line, pattern) != NULL) {
            printf("First occurrence of '%s' found at line %d: %s",
pattern, line_number, line);
            fclose(file);
            return;
        }
        line_number++;
    }
```

```
    printf("Pattern '%s' not found in the file.\n", pattern);
    fclose(file);
}

void search_all(const char *filename, const char *pattern) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return;
    }

    char line[MAX];
    int line_number = 1;
    int found = 0;

    while (fgets(line, sizeof(line), file) != NULL) {
        if (strstr(line, pattern) != NULL) {
            printf("Occurrence of '%s' found at line %d: %s", pattern,
line_number, line);
            found = 1;
        }
        line_number++;
    }

    if (!found) {
        printf("Pattern '%s' not found in the file.\n", pattern);
    }

    fclose(file);
}

int main() {
    char input[MAX];
    char *args[MAX];

    while (1) {
        printf("myshell$ ");
        fgets(input, MAX, stdin);
        input[strcspn(input, "\n")] = 0;  // Remove newline character
```

```
        int i = 0;
        args[i] = strtok(input, " ");
        while (args[i] != NULL) {
            i++;
            args[i] = strtok(NULL, " ");
        }

        if (args[0] == NULL) {
            continue;  // Empty command
        }

        if (strcmp(args[0], "exit") == 0) {
            break;  // Exit the shell
        } else if (strcmp(args[0], "search") == 0 && args[1] != NULL &&
args[2] != NULL) {
            if (strcmp(args[1], "f") == 0) {
                search_first(args[2], args[3]);
            } else if (strcmp(args[1], "a") == 0) {
                search_all(args[2], args[3]);
            } else {
                printf("Invalid search option. Use 'f' for first
occurrence or 'a' for all occurrences.\n");
            }
        } else {
            pid_t pid = fork();
            if (pid == 0) {
                execvp(args[0], args);
                perror("Error executing command");
                exit(1);
            } else {
                wait(NULL);
            }
        }
    }

    return 0;
}
```

Slip no-7

Q.1 Write the simulation program for demand paging and show the page scheduling and total number of page faults according the Optimal page replacement algorithm. Assume the memory of n frames.
Reference String : 7, 5, 4, 8, 5, 7, 2, 3, 1, 3, 5, 9, 4, 6, 2

```c
#include <stdio.h>

#define MAX 100

int findOptimal(int frames[], int n, int ref[], int currentIndex, int
ref_len) {
    int optimalIndex = -1, farthest = currentIndex;

    for (int i = 0; i < n; i++) {
        int j;
        for (j = currentIndex; j < ref_len; j++) {
            if (frames[i] == ref[j]) {
                if (j > farthest) {
                    farthest = j;
                    optimalIndex = i;
                }
                break;
            }
        }
        if (j == ref_len) {
            return i; // This frame is not going to be used again
        }
    }
    return (optimalIndex != -1) ? optimalIndex : 0; // Return the index to
replace
}

int main() {
    int frames[MAX], ref[MAX], page_faults = 0, n, ref_len;

    printf("Enter the number of frames: ");
    scanf("%d", &n);
```

```c
printf("Enter the number of pages in the reference string: ");
scanf("%d", &ref_len);

printf("Enter the reference string: ");
for (int i = 0; i < ref_len; i++) {
    scanf("%d", &ref[i]);
}

for (int i = 0; i < n; i++) {
    frames[i] = -1; // Initialize frames to -1 (empty)
}

for (int i = 0; i < ref_len; i++) {
    int page = ref[i], found = 0;

    for (int j = 0; j < n; j++) {
        if (frames[j] == page) {
            found = 1; // Page hit
            break;
        }
    }

    if (!found) {
        if (i < n) {
            frames[i] = page; // Fill empty frames
        } else {
            int index = findOptimal(frames, n, ref, i + 1, ref_len);
            frames[index] = page; // Replace the page
        }
        page_faults++;

        printf("Page %d loaded. Current frames: ", page);
        for (int j = 0; j < n; j++) {
            if (frames[j] != -1) {
                printf("%d ", frames[j]);
            } else {
                printf("- ");
            }
        }
```

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

    printf("Total page faults: %d\n", page_faults);

    return 0;
}
```

Q.2 Write a program to implement shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following commands.
myshell$ search a filename pattern :- To search all the occurrence of pattern in the file.
myshell$ search c filename pattern :- To count the number of occurrence of pattern in the file.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

#define MAX 1024

void search_all(const char *filename, const char *pattern) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return;
    }

    char line[MAX];
    int line_number = 1;
    int found = 0;

    while (fgets(line, sizeof(line), file) != NULL) {
```

```
        if (strstr(line, pattern) != NULL) {
            printf("Occurrence of '%s' found at line %d: %s", pattern,
line_number, line);
            found = 1;
        }
        line_number++;
    }

    if (!found) {
        printf("Pattern '%s' not found in the file.\n", pattern);
    }

    fclose(file);
}

void count_occurrences(const char *filename, const char *pattern) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return;
    }

    char line[MAX];
    int count = 0;

    while (fgets(line, sizeof(line), file) != NULL) {
        char *ptr = line;
        while ((ptr = strstr(ptr, pattern)) != NULL) {
            count++;
            ptr++;
        }
    }

    printf("Pattern '%s' found %d times in the file.\n", pattern, count);
    fclose(file);
}

int main() {
    char input[MAX];
    char *args[MAX];
```

```
    while (1) {
        printf("myshell$ ");
        fgets(input, MAX, stdin);
        input[strcspn(input, "\n")] = 0; // Remove newline character

        int i = 0;
        args[i] = strtok(input, " ");
        while (args[i] != NULL) {
            i++;
            args[i] = strtok(NULL, " ");
        }

        if (args[0] == NULL) {
            continue; // Empty command
        }

        if (strcmp(args[0], "exit") == 0) {
            break; // Exit the shell
        } else if (strcmp(args[0], "search") == 0 && args[1] != NULL &&
args[2] != NULL) {
            if (strcmp(args[1], "a") == 0) {
                search_all(args[2], args[3]);
            } else if (strcmp(args[1], "c") == 0) {
                count_occurrences(args[2], args[3]);
            } else {
                printf("Invalid search option. Use 'a' for all occurrences
or 'c' for count.\n");
            }
        } else {
            pid_t pid = fork();
            if (pid == 0) {
                execvp(args[0], args);
                perror("Error executing command");
                exit(1);
            } else {
                wait(NULL);
            }
        }
    }
```

```
    return 0;
}
```

Slip no-8

Q.1 Write the simulation program for demand paging and show the page scheduling and total number of page faults according the LRU page replacement algorithm. Assume the memory of n frames.
Reference String : 8, 5, 7, 8, 5, 7, 2, 3, 7, 3, 5, 9, 4, 6, 2

```c
#include <stdio.h>

#define MAX 100

int findLRU(int frames[], int n, int time[], int currentIndex) {
    int lruIndex = 0, minTime = time[0];
    for (int i = 1; i < n; i++) {
        if (time[i] < minTime) {
            minTime = time[i];
            lruIndex = i;
        }
    }
    return lruIndex;
}

int main() {
    int frames[MAX], ref[MAX], time[MAX];
    int page_faults = 0, n, ref_len;

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

    printf("Enter the number of pages in the reference string: ");
    scanf("%d", &ref_len);

    printf("Enter the reference string: ");
```

```
    for (int i = 0; i < ref_len; i++) {
        scanf("%d", &ref[i]);
    }

    for (int i = 0; i < n; i++) {
        frames[i] = -1; // Initialize frames to -1 (empty)
        time[i] = 0;    // Initialize time for LRU tracking
    }

    for (int i = 0; i < ref_len; i++) {
        int page = ref[i], found = 0;

        for (int j = 0; j < n; j++) {
            if (frames[j] == page) {
                found = 1; // Page hit
                time[j] = i; // Update the time of the page
                break;
            }
        }

        if (!found) {
            int lruIndex;
            for (int j = 0; j < n; j++) {
                if (frames[j] == -1) {
                    lruIndex = j; // Empty frame found
                    break;
                }
                if (j == n - 1) {
                    lruIndex = findLRU(frames, n, time, i);
                }
            }
            frames[lruIndex] = page; // Replace the page
            time[lruIndex] = i;      // Update the time for the replaced
page
            page_faults++;

            printf("Page %d loaded. Current frames: ", page);
            for (int j = 0; j < n; j++) {
                if (frames[j] != -1) {
                    printf("%d ", frames[j]);
```

```
                } else {
                    printf("- ");
                }
            }
            printf("\n");
        }
    }

    printf("Total page faults: %d\n", page_faults);

    return 0;
}
```

Q.2 Write a programto implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following commands.
myshell$ search f filename pattern :- To display first occurrence of pattern in the file.
myshell$ search c filename pattern :- To count the number of occurrence of pattern in the file.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

#define MAX 1024

void search_first(const char *filename, const char *pattern) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return;
    }

    char line[MAX];
```

```
    int line_number = 1;

    while (fgets(line, sizeof(line), file) != NULL) {
        if (strstr(line, pattern) != NULL) {
            printf("First occurrence of '%s' found at line %d: %s",
pattern, line_number, line);
            fclose(file);
            return;
        }
        line_number++;
    }

    printf("Pattern '%s' not found in the file.\n", pattern);
    fclose(file);
}

void count_occurrences(const char *filename, const char *pattern) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return;
    }

    char line[MAX];
    int count = 0;

    while (fgets(line, sizeof(line), file) != NULL) {
        char *ptr = line;
        while ((ptr = strstr(ptr, pattern)) != NULL) {
            count++;
            ptr++;
        }
    }

    printf("Pattern '%s' found %d times in the file.\n", pattern, count);
    fclose(file);
}

int main() {
    char input[MAX];
```

```
    char *args[MAX];

    while (1) {
        printf("myshell$ ");
        fgets(input, MAX, stdin);
        input[strcspn(input, "\n")] = 0;

        int i = 0;
        args[i] = strtok(input, " ");
        while (args[i] != NULL) {
            i++;
            args[i] = strtok(NULL, " ");
        }

        if (args[0] == NULL) {
            continue;
        }

        if (strcmp(args[0], "exit") == 0) {
            break;
        } else if (strcmp(args[0], "search") == 0 && args[1] != NULL &&
args[2] != NULL) {
            if (strcmp(args[1], "f") == 0) {
                search_first(args[2], args[3]);
            } else if (strcmp(args[1], "c") == 0) {
                count_occurrences(args[2], args[3]);
            } else {
                printf("Invalid search option. Use 'f' for first
occurrence or 'c' for count.\n");
            }
        } else {
            pid_t pid = fork();
            if (pid == 0) {
                execvp(args[0], args);
                perror("Error executing command");
                exit(1);
            } else {
                wait(NULL);
            }
        }
```

```
    }

    return 0;
}
```

Slip no-9

Q.1 Write the simulation program for demand paging and show the page scheduling and total number of page faults according the FIFO page replacement algorithm. Assume the memory of n frames.
Reference String : 8, 5, 7, 8, 5, 7, 2, 3, 7, 3, 5, 9, 4, 6, 2

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

#define MAX 100

int main() {
    int frames[MAX], ref[MAX], n, ref_len;
    int page_faults = 0, front = 0, rear = 0;

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

    printf("Enter the number of pages in the reference string: ");
    scanf("%d", &ref_len);

    printf("Enter the reference string: ");
    for (int i = 0; i < ref_len; i++) {
        scanf("%d", &ref[i]);
    }

    for (int i = 0; i < n; i++) {
        frames[i] = -1;
    }

    for (int i = 0; i < ref_len; i++) {
        int page = ref[i];
```

```
        int found = 0;

        for (int j = 0; j < n; j++) {
            if (frames[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            frames[rear] = page;
            rear = (rear + 1) % n;
            if (front == rear) {
                front = (front + 1) % n;
            }
            page_faults++;

            printf("Page %d loaded. Current frames: ", page);
            for (int j = 0; j < n; j++) {
                if (frames[j] != -1) {
                    printf("%d ", frames[j]);
                } else {
                    printf("- ");
                }
            }
            printf("\n");
        }
    }

    printf("Total page faults: %d\n", page_faults);

    return 0;
}
```

Q.2 Write a program to implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following commands.

myshell$ search f filename pattern :- To display first occurrence of pattern in the file.
myshell$ search a filename pattern :- To search all the occurrence of pattern in the file.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

#define MAX 1024

void search_first(const char *filename, const char *pattern) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return;
    }

    char line[MAX];
    int line_number = 1;

    while (fgets(line, sizeof(line), file) != NULL) {
        if (strstr(line, pattern) != NULL) {
            printf("First occurrence of '%s' found at line %d: %s",
pattern, line_number, line);
            fclose(file);
            return;
        }
        line_number++;
    }

    printf("Pattern '%s' not found in the file.\n", pattern);
    fclose(file);
}

void search_all(const char *filename, const char *pattern) {
    FILE *file = fopen(filename, "r");
```

```
    if (file == NULL) {
        perror("Error opening file");
        return;
    }

    char line[MAX];
    int line_number = 1;
    int found = 0;

    while (fgets(line, sizeof(line), file) != NULL) {
        if (strstr(line, pattern) != NULL) {
            printf("Occurrence of '%s' found at line %d: %s", pattern,
line_number, line);
            found = 1;
        }
        line_number++;
    }

    if (!found) {
        printf("Pattern '%s' not found in the file.\n", pattern);
    }

    fclose(file);
}

int main() {
    char input[MAX];
    char *args[MAX];

    while (1) {
        printf("myshell$ ");
        fgets(input, MAX, stdin);
        input[strcspn(input, "\n")] = 0;

        int i = 0;
        args[i] = strtok(input, " ");
        while (args[i] != NULL) {
            i++;
            args[i] = strtok(NULL, " ");
        }
```

```
        if (args[0] == NULL) {
            continue;
        }

        if (strcmp(args[0], "exit") == 0) {
            break;
        } else if (strcmp(args[0], "search") == 0 && args[1] != NULL &&
args[2] != NULL) {
            if (strcmp(args[1], "f") == 0) {
                search_first(args[2], args[3]);
            } else if (strcmp(args[1], "a") == 0) {
                search_all(args[2], args[3]);
            } else {
                printf("Invalid search option. Use 'f' for first
occurrence or 'a' for all occurrences.\n");
            }
        } else {
            pid_t pid = fork();
            if (pid == 0) {
                execvp(args[0], args);
                perror("Error executing command");
                exit(1);
            } else {
                wait(NULL);
            }
        }
    }

    return 0;
}
```

Slip no-10

Q.1 Write the simulation program for demand paging and show the page scheduling and total number of page faults according the FIFO page replacement algorithm. Assume the memory of n frames.
Reference String : 2, 4, 5, 6, 9, 4, 7, 3, 4, 5, 6, 7, 2, 4, 7, 1

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

#define MAX 100

int main() {
    int frames[MAX], ref[MAX], n, ref_len;
    int page_faults = 0, front = 0, rear = 0;

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

    printf("Enter the number of pages in the reference string: ");
    scanf("%d", &ref_len);

    printf("Enter the reference string: ");
    for (int i = 0; i < ref_len; i++) {
        scanf("%d", &ref[i]);
    }

    for (int i = 0; i < n; i++) {
        frames[i] = -1;  // Initialize frames to -1 (empty)
    }

    for (int i = 0; i < ref_len; i++) {
        int page = ref[i];
        int found = 0;

        // Check if the page is already in the frames
        for (int j = 0; j < n; j++) {
            if (frames[j] == page) {
                found = 1;
                break;
            }
        }

        // Page fault occurred
        if (!found) {
```

```
            frames[rear] = page;  // Add the new page to the frames
            rear = (rear + 1) % n; // Move to the next frame
            if (front == rear) {
                front = (front + 1) % n; // Adjust front if frames are
full
            }
            page_faults++;

            // Print current state of frames
            printf("Page %d loaded. Current frames: ", page);
            for (int j = 0; j < n; j++) {
                if (frames[j] != -1) {
                    printf("%d ", frames[j]);
                } else {
                    printf("- ");
                }
            }
            printf("\n");
        }
    }

    printf("Total page faults: %d\n", page_faults);

    return 0;
}
```

Q.2 Write a program to implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following ‘list’ commands as
myshell$ list f dirname :- To print names of all the files in current directory.
myshell$ list i dirname :- To print names and inodes of the files in the current directory.

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

```c
#include <sys/stat.h>
#include <unistd.h>

#define MAX 1024

void list_files(const char *dirname, int show_inodes) {
    struct dirent *entry;
    struct stat file_stat;
    char filepath[MAX];

    DIR *dir = opendir(dirname);
    if (dir == NULL) {
        perror("Error opening directory");
        return;
    }

    while ((entry = readdir(dir)) != NULL) {
        if (entry->d_name[0] != '.') { // Skip hidden files
            snprintf(filepath, sizeof(filepath), "%s/%s", dirname,
entry->d_name);
            if (show_inodes) {
                stat(filepath, &file_stat);
                printf("Inode: %lu - Name: %s\n", file_stat.st_ino,
entry->d_name);
            } else {
                printf("%s\n", entry->d_name);
            }
        }
    }
    closedir(dir);
}

int main() {
    char input[MAX];
    char *args[MAX];

    while (1) {
        printf("myshell$ ");
        fgets(input, MAX, stdin);
        input[strcspn(input, "\n")] = 0;
```

```
        int i = 0;
        args[i] = strtok(input, " ");
        while (args[i] != NULL) {
            i++;
            args[i] = strtok(NULL, " ");
        }

        if (args[0] == NULL) {
            continue;
        }

        if (strcmp(args[0], "exit") == 0) {
            break;
        } else if (strcmp(args[0], "list") == 0 && args[1] != NULL &&
args[2] != NULL) {
            if (strcmp(args[1], "f") == 0) {
                list_files(args[2], 0); // List files
            } else if (strcmp(args[1], "i") == 0) {
                list_files(args[2], 1); // List files with inodes
            } else {
                printf("Invalid list option. Use 'f' for filenames or 'i'
for inodes.\n");
            }
        } else {
            pid_t pid = fork();
            if (pid == 0) {
                execvp(args[0], args);
                perror("Error executing command");
                exit(1);
            } else {
                wait(NULL);
            }
        }
    }

    return 0;
}
```

Slip no-11

Q.1 Write the simulation program for demand paging and show the page scheduling and total number of page faults according the LFU page replacement algorithm. Assume the memory of n frames.
Reference String : 3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6

```c
#include <stdio.h>

#define MAX_FRAMES 10
#define MAX_REFERENCES 100

// Function to find the least frequently used page
int findLFU(int frames[], int freq[], int n) {
    int minFreq = freq[0], minIndex = 0;
    for (int i = 1; i < n; i++) {
        if (freq[i] < minFreq) {
            minFreq = freq[i];
            minIndex = i;
        }
    }
    return minIndex;
}

int main() {
    int n, referenceString[MAX_REFERENCES], frames[MAX_FRAMES],
freq[MAX_FRAMES];
    int pageFaults = 0, found, index;

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

    printf("Enter reference string (-1 to stop): ");
    int i = 0;
    while (1) {
        int page;
        scanf("%d", &page);
        if (page == -1) break;
        referenceString[i++] = page;
    }
```

```
    int refCount = i;

    // Initialize frames and frequency arrays
    for (i = 0; i < n; i++) {
        frames[i] = -1;
        freq[i] = 0;
    }

    // Simulate LFU page replacement
    for (i = 0; i < refCount; i++) {
        int page = referenceString[i];
        found = 0;

        // Check if the page is already in a frame
        for (int j = 0; j < n; j++) {
            if (frames[j] == page) {
                found = 1;
                freq[j]++;  // Increment frequency
                break;
            }
        }

        if (!found) {
            pageFaults++;

            // Find a free frame or replace LFU page
            if (frames[n - 1] == -1) {
                for (int j = 0; j < n; j++) {
                    if (frames[j] == -1) {
                        frames[j] = page;
                        freq[j] = 1;
                        break;
                    }
                }
            } else {
                index = findLFU(frames, freq, n);  // Find the least
frequently used page
                frames[index] = page;              // Replace it with the
new page
                freq[index] = 1;
```

```
            }
        }

        // Display current frames status
        printf("\nFrames: ");
        for (int j = 0; j < n; j++) {
            if (frames[j] != -1)
                printf("%d ", frames[j]);
            else
                printf("- ");
        }
    }

    printf("\n\nTotal page faults: %d\n", pageFaults);

    return 0;
}
```

Q.2 Write a C program to implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following ‘list’ commands as
myshell$ list f dirname :- To print names of all the files in current directory.
myshell$ list n dirname :- To print the number of all entries in the current directory

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <dirent.h>

#define MAX_INPUT 100
#define MAX_ARGS 10

void list_files(const char *dirname) {
```

```
    struct dirent *entry;
    DIR *dp = opendir(dirname);

    if (dp == NULL) {
        perror("opendir");
        return;
    }

    while ((entry = readdir(dp)) != NULL) {
        if (entry->d_name[0] != '.') { // Skip hidden files
            printf("%s\n", entry->d_name);
        }
    }

    closedir(dp);
}

void list_count(const char *dirname) {
    struct dirent *entry;
    DIR *dp = opendir(dirname);
    int count = 0;

    if (dp == NULL) {
        perror("opendir");
        return;
    }

    while ((entry = readdir(dp)) != NULL) {
        if (entry->d_name[0] != '.') { // Skip hidden files
            count++;
        }
    }

    closedir(dp);
    printf("Total entries: %d\n", count);
}

int main() {
    char input[MAX_INPUT];
    char *args[MAX_ARGS];
```

```
    char *token;

    while (1) {
        printf("myshell$ ");
        fgets(input, sizeof(input), stdin); // Read input

        // Remove newline character at the end
        input[strcspn(input, "\n")] = 0;

        // Tokenize the input
        int i = 0;
        token = strtok(input, " ");
        while (token != NULL && i < MAX_ARGS - 1) {
            args[i++] = token;
            token = strtok(NULL, " ");
        }
        args[i] = NULL; // Null terminate the argument list

        // Exit command
        if (args[0] != NULL && strcmp(args[0], "exit") == 0) {
            break;
        }

        // Handle "list" command
        if (args[0] != NULL && strcmp(args[0], "list") == 0) {
            if (args[1] != NULL && args[2] != NULL) {
                if (strcmp(args[1], "f") == 0) {
                    list_files(args[2]);
                } else if (strcmp(args[1], "n") == 0) {
                    list_count(args[2]);
                } else {
                    printf("Invalid option for list command\n");
                }
            } else {
                printf("Usage: list <f|n> <dirname>\n");
            }
            continue;
        }

        // Fork and execute other commands
```

```
        pid_t pid = fork();
        if (pid == 0) {  // Child process
            if (execvp(args[0], args) == -1) {
                perror("execvp");
            }
            exit(EXIT_FAILURE);
        } else if (pid < 0) {  // Fork failed
            perror("fork");
        } else {  // Parent process
            wait(NULL);  // Wait for child to finish
        }
    }

    return 0;
}
```

Slip no-12

Q.1 Write the simulation program for demand paging and show the page scheduling and total number of page faults according the LRU page replacement algorithm. Assume the memory of n frames.
Reference String : 3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6

```
#include <stdio.h>

#define MAX_FRAMES 10
#define MAX_REFERENCES 20

// Function to find the least recently used page
int findLRU(int time[], int n) {
    int i, min = time[0], pos = 0;
    for (i = 1; i < n; i++) {
        if (time[i] < min) {
            min = time[i];
            pos = i;
        }
    }
    return pos;
```

```
}

int main() {
    int n, frames[MAX_FRAMES], referenceString[MAX_REFERENCES],
time[MAX_FRAMES];
    int pageFaults = 0, counter = 0;
    int i, j, pos, flag1, flag2;

    // Define the number of frames
    printf("Enter the number of frames: ");
    scanf("%d", &n);

    // Initialize frames
    for (i = 0; i < n; i++) {
        frames[i] = -1;
    }

    // Define the reference string
    int referenceStringLength = 15;
    int referenceStringInput[] = {3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2,
4, 6};

    printf("Reference String: ");
    for (i = 0; i < referenceStringLength; i++) {
        referenceString[i] = referenceStringInput[i];
        printf("%d ", referenceString[i]);
    }
    printf("\n");

    // Simulate LRU page replacement
    for (i = 0; i < referenceStringLength; i++) {
        flag1 = flag2 = 0;

        // Check if the page is already in a frame
        for (j = 0; j < n; j++) {
            if (frames[j] == referenceString[i]) {
                counter++;
                time[j] = counter; // Update time for LRU
                flag1 = flag2 = 1;
                break;
```

```
            }
        }

        // If page is not in a frame (page fault occurs)
        if (flag1 == 0) {
            for (j = 0; j < n; j++) {
                if (frames[j] == -1) {  // Empty frame available
                    counter++;
                    pageFaults++;
                    frames[j] = referenceString[i];
                    time[j] = counter;
                    flag2 = 1;
                    break;
                }
            }
        }

        // Replace the least recently used page
        if (flag2 == 0) {
            pos = findLRU(time, n);
            counter++;
            pageFaults++;
            frames[pos] = referenceString[i];
            time[pos] = counter;
        }

        // Display current frames status
        printf("\nFrames: ");
        for (j = 0; j < n; j++) {
            if (frames[j] != -1)
                printf("%d ", frames[j]);
            else
                printf("- ");
        }
    }

    printf("\n\nTotal number of page faults: %d\n", pageFaults);

    return 0;
}
```

Q.2 Write a program to implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following ‘list’ commands as
myshell$ list f dirname :- To print names of all the files in current directory.
myshell$ list n dirname :- To print the number of all entries in the current directory

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <dirent.h>

#define MAX_INPUT 100
#define MAX_ARGS 10

void list_files(const char *dirname) {
    struct dirent *entry;
    DIR *dp = opendir(dirname);

    if (dp == NULL) return;

    while ((entry = readdir(dp)) != NULL) {
        if (entry->d_name[0] != '.') {
            printf("%s\n", entry->d_name);
        }
    }

    closedir(dp);
}

void list_count(const char *dirname) {
    struct dirent *entry;
    DIR *dp = opendir(dirname);
    int count = 0;
```

```
    if (dp == NULL) return;

    while ((entry = readdir(dp)) != NULL) {
        if (entry->d_name[0] != '.') {
            count++;
        }
    }

    closedir(dp);
    printf("%d\n", count);
}

int main() {
    char input[MAX_INPUT];
    char *args[MAX_ARGS];
    char *token;

    while (1) {
        printf("myshell$ ");
        fgets(input, sizeof(input), stdin);
        input[strcspn(input, "\n")] = 0;

        int i = 0;
        token = strtok(input, " ");
        while (token != NULL && i < MAX_ARGS - 1) {
            args[i++] = token;
            token = strtok(NULL, " ");
        }
        args[i] = NULL;

        if (args[0] == NULL) continue;

        if (strcmp(args[0], "exit") == 0) break;

        if (strcmp(args[0], "list") == 0 && args[1] && args[2]) {
            if (strcmp(args[1], "f") == 0) {
                list_files(args[2]);
            } else if (strcmp(args[1], "n") == 0) {
                list_count(args[2]);
```

```
            }
            continue;
        }

        pid_t pid = fork();
        if (pid == 0) {
            if (execvp(args[0], args) == -1) perror("execvp");
            exit(EXIT_FAILURE);
        } else if (pid > 0) {
            wait(NULL);
        }
    }

    return 0;
}
```

Slip no-13

Q.1 Write a C program to implement the shell which displays the command prompt “myshell$”. It accepts the command, tokenize the command line and execute it by creating the child process. Also implement the additional command ‘typeline’ as
typeline -a filename :- To print all lines in the file.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <fcntl.h>

#define MAX_INPUT 100
#define MAX_ARGS 10

// Function to print all lines from the given file
void typeline_all(const char *filename) {
    FILE *file = fopen(filename, "r");
```

```
    char line[256];

    if (file == NULL) {
        perror("Error opening file");
        return;
    }

    // Read and print each line from the file
    while (fgets(line, sizeof(line), file)) {
        printf("%s", line);
    }

    fclose(file);
}

int main() {
    char input[MAX_INPUT];
    char *args[MAX_ARGS];
    char *token;

    while (1) {
        // Display the prompt
        printf("myshell$ ");
        // Get input from the user
        fgets(input, sizeof(input), stdin);
        // Remove newline character from the input
        input[strcspn(input, "\n")] = 0;

        // Tokenize the input
        int i = 0;
        token = strtok(input, " ");
        while (token != NULL && i < MAX_ARGS - 1) {
            args[i++] = token;
            token = strtok(NULL, " ");
        }
        args[i] = NULL;

        // If no input is given, continue
        if (args[0] == NULL) continue;
```

```
        // Exit command
        if (strcmp(args[0], "exit") == 0) {
            break;
        }

        // Custom 'typeline' command implementation
        if (strcmp(args[0], "typeline") == 0) {
            if (args[1] != NULL && strcmp(args[1], "-a") == 0 && args[2]
!= NULL) {
                // Call function to print all lines in the file
                typeline_all(args[2]);
            } else {
                printf("Usage: typeline -a filename\n");
            }
            continue;
        }

        // Fork a child process to execute other commands
        pid_t pid = fork();
        if (pid == 0) {  // Child process
            if (execvp(args[0], args) == -1) {
                perror("Error executing command");
            }
            exit(EXIT_FAILURE);
        } else if (pid > 0) {  // Parent process
            wait(NULL);  // Wait for the child process to complete
        } else {
            perror("Fork failed");
        }
    }

    return 0;
}
```

Q.2 Write the simulation program for Round Robin scheduling for given time quantum. The arrival time and first CPU-burst of different jobs should be input to the system. Accept no. of Processes, arrival time and burst time. The output should give the Gantt chart, turnaround time and waiting time for each

process. Also display the average turnaround time and average waiting time.

```c
#include <stdio.h>

int main() {
    int n, i, time = 0, remain, flag = 0, time_quantum;
    int waiting_time = 0, turnaround_time = 0;

    printf("Enter the number of processes: ");
    scanf("%d", &n);

    int at[n], bt[n], rt[n]; // Arrival time, burst time, remaining time
    for (i = 0; i < n; i++) {
        printf("Enter arrival time and burst time for process P%d: ",
i+1);
        scanf("%d %d", &at[i], &bt[i]);
        rt[i] = bt[i];
    }

    printf("Enter the time quantum: ");
    scanf("%d", &time_quantum);

    remain = n; // Number of processes remaining
    printf("\nGantt Chart:\n");
    while (remain != 0) {
        for (i = 0; i < n; i++) {
            if (rt[i] > 0 && at[i] <= time) {
                if (rt[i] <= time_quantum) {
                    time += rt[i];
                    printf("| P%d (%d) ", i+1, time);
                    rt[i] = 0;
                    remain--;
                    waiting_time += time - at[i] - bt[i];
                    turnaround_time += time - at[i];
                } else {
                    rt[i] -= time_quantum;
                    time += time_quantum;
                    printf("| P%d (%d) ", i+1, time);
                }
            }
```

```
        }
    }

    printf("|\n");

    printf("\nProcess\tTurnaround Time\tWaiting Time\n");
    for (i = 0; i < n; i++) {
        int turn_time = (time - at[i]);
        int wait_time = (turn_time - bt[i]);
        printf("P%d\t%d\t\t%d\n", i+1, turn_time, wait_time);
    }

    printf("\nAverage Turnaround Time: %.2f", (float)turnaround_time / n);
    printf("\nAverage Waiting Time: %.2f", (float)waiting_time / n);

    return 0;
}
```

Slip no-14

Q.1 Write a C program to implement the shell which displays the command prompt “myshell$”. It accepts the command, tokenize the command line and execute it by creating the child process. Also implement the additional command ‘typeline’ as
typeline +n filename :- To print first n lines in the file.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

#define MAX_INPUT 100
#define MAX_ARGS 10

// Function to print first 'n' lines from the file
void typeline_n(const char *filename, int n) {
    FILE *file = fopen(filename, "r");
```

```
    char line[256];
    int count = 0;

    if (file == NULL) {
        perror("Error opening file");
        return;
    }

    // Read and print the first 'n' lines from the file
    while (fgets(line, sizeof(line), file) && count < n) {
        printf("%s", line);
        count++;
    }

    fclose(file);
}

int main() {
    char input[MAX_INPUT];
    char *args[MAX_ARGS];
    char *token;

    while (1) {
        // Display the prompt
        printf("myshell$ ");
        // Get input from the user
        fgets(input, sizeof(input), stdin);
        // Remove newline character from the input
        input[strcspn(input, "\n")] = 0;

        // Tokenize the input
        int i = 0;
        token = strtok(input, " ");
        while (token != NULL && i < MAX_ARGS - 1) {
            args[i++] = token;
            token = strtok(NULL, " ");
        }
        args[i] = NULL;

        // If no input is given, continue
```

```
        if (args[0] == NULL) continue;

        // Exit command
        if (strcmp(args[0], "exit") == 0) {
            break;
        }

        // Custom 'typeline' command implementation
        if (strcmp(args[0], "typeline") == 0) {
            if (args[1] != NULL && args[1][0] == '+' && args[2] != NULL) {
                int n = atoi(&args[1][1]);  // Extract number from +n
                typeline_n(args[2], n);     // Call function to print
first 'n' lines
            } else {
                printf("Usage: typeline +n filename\n");
            }
            continue;
        }

        // Fork a child process to execute other commands
        pid_t pid = fork();
        if (pid == 0) {  // Child process
            if (execvp(args[0], args) == -1) {
                perror("Error executing command");
            }
            exit(EXIT_FAILURE);
        } else if (pid > 0) {  // Parent process
            wait(NULL);  // Wait for the child process to complete
        } else {
            perror("Fork failed");
        }
    }

    return 0;
}
```

Q.2 Write a C program to simulate Non-preemptive Shortest Job First (SJF) – scheduling. The arrival time and first CPU-burst of different jobs should be

input to the system. Accept no. of Processes, arrival time and burst time. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time

```c
#include <stdio.h>

int main() {
    int n, i, j, time = 0;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    int at[n], bt[n], wt[n], tat[n], process[n];
    int completed = 0, total_wt = 0, total_tat = 0, min_bt, index;
    int finished[n], current_time = 0;

    for (i = 0; i < n; i++) {
        printf("Enter arrival time and burst time for process P%d: ", i +
1);
        scanf("%d %d", &at[i], &bt[i]);
        process[i] = i + 1;
        finished[i] = 0;
    }

    printf("\nGantt Chart:\n");
    while (completed < n) {
        min_bt = 9999;
        index = -1;

        for (i = 0; i < n; i++) {
            if (at[i] <= current_time && !finished[i] && bt[i] < min_bt) {
                min_bt = bt[i];
                index = i;
            }
        }

        if (index != -1) {
            current_time += bt[index];
            printf("| P%d (%d) ", process[index], current_time);
            tat[index] = current_time - at[index];
            wt[index] = tat[index] - bt[index];
```

```
            total_tat += tat[index];
            total_wt += wt[index];
            finished[index] = 1;
            completed++;
        } else {
            current_time++;
        }
    }
    printf("|\n");

    printf("\nProcess\tTurnaround Time\tWaiting Time\n");
    for (i = 0; i < n; i++) {
        printf("P%d\t%d\t\t%d\n", process[i], tat[i], wt[i]);
    }

    printf("\nAverage Turnaround Time: %.2f", (float)total_tat / n);
    printf("\nAverage Waiting Time: %.2f\n", (float)total_wt / n);

    return 0;
}
```

Slip no-15

Q.1 Write a C program to implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following ‘list’ commands as
myshell$ list f dirname :- To print names of all the files in current directory.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <dirent.h>

#define MAX_INPUT 100
```

```c
#define MAX_ARGS 10

// Function to list all files in the directory
void list_files(const char *dirname) {
    struct dirent *de;
    DIR *dr = opendir(dirname);

    if (dr == NULL) {
        perror("Could not open directory");
        return;
    }

    while ((de = readdir(dr)) != NULL) {
        if (de->d_type == DT_REG) {
            printf("%s\n", de->d_name);
        }
    }

    closedir(dr);
}

int main() {
    char input[MAX_INPUT];
    char *args[MAX_ARGS];
    char *token;

    while (1) {
        printf("myshell$ ");
        fgets(input, sizeof(input), stdin);
        input[strcspn(input, "\n")] = 0;

        int i = 0;
        token = strtok(input, " ");
        while (token != NULL && i < MAX_ARGS - 1) {
            args[i++] = token;
            token = strtok(NULL, " ");
        }
        args[i] = NULL;

        if (args[0] == NULL) continue;
```

```
        if (strcmp(args[0], "exit") == 0) {
            break;
        }

        if (strcmp(args[0], "list") == 0 && strcmp(args[1], "f") == 0 &&
args[2] != NULL) {
            list_files(args[2]);
            continue;
        }

        pid_t pid = fork();
        if (pid == 0) {
            if (execvp(args[0], args) == -1) {
                perror("Error executing command");
            }
            exit(EXIT_FAILURE);
        } else if (pid > 0) {
            wait(NULL);
        } else {
            perror("Fork failed");
        }
    }

    return 0;
}
```

Q.2 Write the program to simulate preemptive Shortest Job First (SJF) – scheduling. The arrival time and first CPU-burst of different jobs should be input to the system. Accept no. of Processes, arrival time and burst time. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time

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

typedef struct {
    int id;
    int arrival_time;
    int burst_time;
```

```
    int remaining_time;
    int waiting_time;
    int turnaround_time;
} Process;

void find_sjf(int n, Process processes[]) {
    int total_time = 0;
    int completed = 0;
    int current_time = 0;
    int min_remaining_time, index;

    while (completed < n) {
        min_remaining_time = 9999;
        index = -1;

        for (int i = 0; i < n; i++) {
            if (processes[i].arrival_time <= current_time &&
processes[i].remaining_time > 0) {
                if (processes[i].remaining_time < min_remaining_time) {
                    min_remaining_time = processes[i].remaining_time;
                    index = i;
                }
            }
        }

        if (index != -1) {
            processes[index].remaining_time--;
            current_time++;

            if (processes[index].remaining_time == 0) {
                processes[index].turnaround_time = current_time -
processes[index].arrival_time;
                processes[index].waiting_time =
processes[index].turnaround_time - processes[index].burst_time;
                completed++;
            }
        } else {
            current_time++;
        }
    }
```

```
}

void print_gantt_chart(int n, Process processes[]) {
    printf("\nGantt Chart:\n|");
    for (int i = 0; i < n; i++) {
        printf(" P%d |", processes[i].id);
    }
    printf("\n");
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    Process processes[n];
    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("Enter arrival time and burst time for process P%d: ", i +
1);
        scanf("%d %d", &processes[i].arrival_time,
&processes[i].burst_time);
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].waiting_time = 0;
        processes[i].turnaround_time = 0;
    }

    find_sjf(n, processes);

    int total_waiting_time = 0;
    int total_turnaround_time = 0;

    printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround
Time\n");
    for (int i = 0; i < n; i++) {
        total_waiting_time += processes[i].waiting_time;
        total_turnaround_time += processes[i].turnaround_time;
        printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id,
processes[i].arrival_time, processes[i].burst_time,
processes[i].waiting_time, processes[i].turnaround_time);
```

```
    }

    printf("\nAverage Waiting Time: %.2f", (float)total_waiting_time / n);
    printf("\nAverage Turnaround Time: %.2f\n",
(float)total_turnaround_time / n);

    print_gantt_chart(n, processes);

    return 0;
}
```

Slip no-16

Q.1 Write a programto implement the toy shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following commands.
count c filename :- To print number of characters in the file.
count w filename :- To print number of words in the file.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define MAX_INPUT 100
#define MAX_ARGS 10

// Function to count characters in a file
void count_characters(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Could not open file");
        return;
    }
```

```
    int count = 0;
    char ch;
    while ((ch = fgetc(file)) != EOF) {
        count++;
    }

    fclose(file);
    printf("Number of characters in '%s': %d\n", filename, count);
}

// Function to count words in a file
void count_words(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Could not open file");
        return;
    }

    int count = 0;
    char word[MAX_INPUT];

    while (fscanf(file, "%s", word) == 1) {
        count++;
    }

    fclose(file);
    printf("Number of words in '%s': %d\n", filename, count);
}

int main() {
    char input[MAX_INPUT];
    char *args[MAX_ARGS];

    while (1) {
        printf("myshell$ ");
        fgets(input, sizeof(input), stdin);
        input[strcspn(input, "\n")] = 0;

        int i = 0;
        char *token = strtok(input, " ");
```

```
        while (token != NULL && i < MAX_ARGS - 1) {
            args[i++] = token;
            token = strtok(NULL, " ");
        }
        args[i] = NULL;

        if (args[0] == NULL) continue;

        if (strcmp(args[0], "exit") == 0) {
            break;
        }

        if (strcmp(args[0], "count") == 0) {
            if (strcmp(args[1], "c") == 0 && args[2] != NULL) {
                count_characters(args[2]);
            } else if (strcmp(args[1], "w") == 0 && args[2] != NULL) {
                count_words(args[2]);
            } else {
                printf("Invalid command syntax.\n");
            }
            continue;
        }

        pid_t pid = fork();
        if (pid == 0) {
            if (execvp(args[0], args) == -1) {
                perror("Error executing command");
            }
            exit(EXIT_FAILURE);
        } else if (pid > 0) {
            wait(NULL);
        } else {
            perror("Fork failed");
        }
    }

    return 0;
}
```

Q.2 Write the program to simulate Non preemptive priority scheduling. The arrival time and first CPU-burst of different jobs should be input to the system. Accept no. of Processes, arrival time and burst time. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

```c
#include <stdio.h>

struct Process {
    int pid;
    int arrival_time;
    int burst_time;
    int priority;
    int completion_time;
    int turnaround_time;
    int waiting_time;
};

// Function to sort processes by arrival time, then priority
void sort_by_arrival_priority(struct Process p[], int n) {
    struct Process temp;
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (p[i].arrival_time > p[j].arrival_time ||
                (p[i].arrival_time == p[j].arrival_time && p[i].priority > p[j].priority)) {
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
    }
}

// Function to simulate Non-preemptive Priority Scheduling
void priority_scheduling(struct Process p[], int n) {
    int current_time = 0;
    int completed = 0;
    int gantt_chart[100][2], gantt_count = 0;
```

```
    while (completed < n) {
        int idx = -1;
        int min_priority = 9999;

        // Select process with the highest priority that has arrived
        for (int i = 0; i < n; i++) {
            if (p[i].arrival_time <= current_time && p[i].completion_time
== 0) {
                if (p[i].priority < min_priority) {
                    min_priority = p[i].priority;
                    idx = i;
                }
            }
        }

        if (idx != -1) {
            gantt_chart[gantt_count][0] = p[idx].pid;
            gantt_chart[gantt_count][1] = current_time;
            gantt_count++;

            current_time += p[idx].burst_time;
            p[idx].completion_time = current_time;
            p[idx].turnaround_time = p[idx].completion_time -
p[idx].arrival_time;
            p[idx].waiting_time = p[idx].turnaround_time -
p[idx].burst_time;
            completed++;
        } else {
            current_time++;
        }
    }

    // Print Gantt Chart
    printf("\nGantt Chart:\n");
    for (int i = 0; i < gantt_count; i++) {
        printf("P%d [%d] -> ", gantt_chart[i][0], gantt_chart[i][1]);
    }
    printf("Finish\n");

    // Display turnaround and waiting time for each process
```

```
    printf("\nProcess\tArrival\tBurst\tPriority\tTAT\tWT\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t\t%d\t%d\n", p[i].pid, p[i].arrival_time,
p[i].burst_time, p[i].priority, p[i].turnaround_time, p[i].waiting_time);
    }
}

// Function to calculate average waiting and turnaround times
void calculate_average(struct Process p[], int n) {
    float avg_turnaround_time = 0.0, avg_waiting_time = 0.0;
    for (int i = 0; i < n; i++) {
        avg_turnaround_time += p[i].turnaround_time;
        avg_waiting_time += p[i].waiting_time;
    }
    printf("\nAverage Turnaround Time: %.2f", avg_turnaround_time / n);
    printf("\nAverage Waiting Time: %.2f\n", avg_waiting_time / n);
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct Process p[n];

    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("Enter arrival time of process P%d: ", i + 1);
        scanf("%d", &p[i].arrival_time);
        printf("Enter burst time of process P%d: ", i + 1);
        scanf("%d", &p[i].burst_time);
        printf("Enter priority of process P%d: ", i + 1);
        scanf("%d", &p[i].priority);
        p[i].completion_time = 0;  // Initialize completion time to 0
    }

    sort_by_arrival_priority(p, n);
    priority_scheduling(p, n);
    calculate_average(p, n);
```

```
    return 0;
}
```

Slip no-17

Q.1 Write the simulation program for demand paging and show the page scheduling and total number of page faults according the Optimal page replacement algorithm. Assume the memory of n frames.
Reference String : 7, 5, 4, 8, 5, 7, 2, 3, 1, 3, 5, 9, 4, 6,

```
#include <stdio.h>

// Function to check if a page is already in a frame
int is_page_in_memory(int page, int frames[], int n) {
    for (int i = 0; i < n; i++) {
        if (frames[i] == page) {
            return 1; // Page found
        }
    }
    return 0; // Page not found
}

// Function to find the optimal page to replace
int find_optimal_page(int reference_string[], int frames[], int n, int
index, int reference_length) {
    int farthest = index;
    int replace_index = -1;

    for (int i = 0; i < n; i++) {
        int j;
        for (j = index; j < reference_length; j++) {
            if (frames[i] == reference_string[j]) {
                if (j > farthest) {
                    farthest = j;
                    replace_index = i;
                }
                break;
            }
```

```
        }
        if (j == reference_length) {
            return i; // If the page is not referenced in the future,
return it for replacement
        }
    }

    return (replace_index == -1) ? 0 : replace_index;
}

// Function to simulate Optimal Page Replacement
void optimal_page_replacement(int reference_string[], int
reference_length, int frames[], int n) {
    int page_faults = 0;
    int index = 0; // To keep track of the current page being referenced

    printf("Page scheduling:\n");

    for (int i = 0; i < reference_length; i++) {
        int current_page = reference_string[i];

        // Check if the page is already in memory
        if (!is_page_in_memory(current_page, frames, n)) {
            page_faults++;

            if (index < n) {
                // If there is still space in memory, simply add the page
                frames[index++] = current_page;
            } else {
                // Find the optimal page to replace
                int replace_index = find_optimal_page(reference_string,
frames, n, i + 1, reference_length);
                frames[replace_index] = current_page;
            }

            // Display the current state of frames
            for (int j = 0; j < n; j++) {
                if (frames[j] != -1) {
                    printf("%d ", frames[j]);
                } else {
```

```
                    printf("- ");
                }
            }
            printf("\n");
        } else {
            // Display the current state if there is no page fault
            for (int j = 0; j < n; j++) {
                if (frames[j] != -1) {
                    printf("%d ", frames[j]);
                } else {
                    printf("- ");
                }
            }
            printf(" (No page fault)\n");
        }
    }

    printf("\nTotal number of page faults: %d\n", page_faults);
}

int main() {
    int reference_string[] = {7, 5, 4, 8, 5, 7, 2, 3, 1, 3, 5, 9, 4, 6};
    int reference_length = sizeof(reference_string) /
sizeof(reference_string[0]);

    int n;
    printf("Enter the number of frames: ");
    scanf("%d", &n);

    int frames[n];
    for (int i = 0; i < n; i++) {
        frames[i] = -1; // Initialize frames with -1 (empty)
    }

    optimal_page_replacement(reference_string, reference_length, frames,
n);

    return 0;
}
```

Q.2 Write the program to simulate FCFS CPU-scheduling. The arrival time and first CPU-burst of different jobs should be input to the system. Accept no. of Processes, arrival time and burst time. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

```
#include <stdio.h>

struct Process {
    int pid;
    int arrival_time;
    int burst_time;
    int completion_time;
    int turnaround_time;
    int waiting_time;
};

void sort_by_arrival(struct Process p[], int n) {
    struct Process temp;
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (p[i].arrival_time > p[j].arrival_time) {
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
    }
}

void fcfs_scheduling(struct Process p[], int n) {
    int current_time = 0;
    printf("\nGantt Chart:\n");
    for (int i = 0; i < n; i++) {
        if (current_time < p[i].arrival_time) {
            current_time = p[i].arrival_time;
        }
        printf("P%d [%d - %d] -> ", p[i].pid, current_time, current_time +
p[i].burst_time);
        p[i].completion_time = current_time + p[i].burst_time;
```

```
        p[i].turnaround_time = p[i].completion_time - p[i].arrival_time;
        p[i].waiting_time = p[i].turnaround_time - p[i].burst_time;
        current_time += p[i].burst_time;
    }
    printf("Finish\n");
}

void calculate_and_display_avg_times(struct Process p[], int n) {
    float total_turnaround_time = 0, total_waiting_time = 0;
    printf("\nProcess\tArrival\tBurst\tCompletion\tTAT\tWT\n");
    for (int i = 0; i < n; i++) {
        total_turnaround_time += p[i].turnaround_time;
        total_waiting_time += p[i].waiting_time;
        printf("P%d\t%d\t%d\t%d\t\t%d\t%d\n", p[i].pid, p[i].arrival_time,
p[i].burst_time,
               p[i].completion_time, p[i].turnaround_time,
p[i].waiting_time);
    }
    printf("\nAverage Turnaround Time: %.2f", total_turnaround_time / n);
    printf("\nAverage Waiting Time: %.2f\n", total_waiting_time / n);
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process p[n];
    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("Enter arrival time for process P%d: ", i + 1);
        scanf("%d", &p[i].arrival_time);
        printf("Enter burst time for process P%d: ", i + 1);
        scanf("%d", &p[i].burst_time);
    }
    sort_by_arrival(p, n);
    fcfs_scheduling(p, n);
    calculate_and_display_avg_times(p, n);
    return 0;
}
```

Slip no-18

Q.1 Write the simulation program for demand paging and show the page scheduling and total number of page faults according the LRU page replacement algorithm. Assume the memory of n frames.
Reference String : 3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6

```c
#include <stdio.h>

int find_lru(int time[], int n) {
    int minimum = time[0], pos = 0;
    for (int i = 1; i < n; ++i) {
        if (time[i] < minimum) {
            minimum = time[i];
            pos = i;
        }
    }
    return pos;
}

void lru_page_replacement(int reference_string[], int reference_length,
int frames[], int n) {
    int time[n], flag1, flag2, pos, page_faults = 0, current_time = 0;

    for (int i = 0; i < n; ++i) {
        frames[i] = -1;  // Initialize frames as empty
    }

    printf("Page scheduling:\n");

    for (int i = 0; i < reference_length; ++i) {
        flag1 = flag2 = 0;

        for (int j = 0; j < n; ++j) {
            if (frames[j] == reference_string[i]) { // Page hit
                flag1 = flag2 = 1;
                time[j] = ++current_time; // Update time for LRU tracking
                break;
```

```
            }
        }

        if (flag1 == 0) {
            for (int j = 0; j < n; ++j) {
                if (frames[j] == -1) { // Empty frame found
                    page_faults++;
                    frames[j] = reference_string[i];
                    time[j] = ++current_time;
                    flag2 = 1;
                    break;
                }
            }
        }

        if (flag2 == 0) { // Page replacement needed
            pos = find_lru(time, n);
            frames[pos] = reference_string[i];
            time[pos] = ++current_time;
            page_faults++;
        }

        // Display current frame status
        for (int j = 0; j < n; ++j) {
            if (frames[j] != -1) {
                printf("%d ", frames[j]);
            } else {
                printf("- ");
            }
        }
        printf("\n");
    }

    printf("\nTotal number of page faults: %d\n", page_faults);
}

int main() {
    int reference_string[] = {3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4,
6};
```

```
    int reference_length = sizeof(reference_string) /
sizeof(reference_string[0]);

    int n;
    printf("Enter the number of frames: ");
    scanf("%d", &n);

    int frames[n];

    lru_page_replacement(reference_string, reference_length, frames, n);

    return 0;
}
```

Q.2 Write a C program to simulate FCFS CPU-scheduling. The arrival time and first CPU-burst of different jobs should be input to the system. Accept no. of Processes, arrival time and burst time. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

```
#include <stdio.h>

struct Process {
    int pid;
    int arrival_time;
    int burst_time;
    int completion_time;
    int turnaround_time;
    int waiting_time;
};

void sort_by_arrival(struct Process p[], int n) {
    struct Process temp;
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (p[i].arrival_time > p[j].arrival_time) {
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
```

```
        }
    }
}

void fcfs_scheduling(struct Process p[], int n) {
    int current_time = 0;
    printf("\nGantt Chart:\n");
    for (int i = 0; i < n; i++) {
        if (current_time < p[i].arrival_time) {
            current_time = p[i].arrival_time;
        }
        printf("P%d [%d - %d] -> ", p[i].pid, current_time, current_time +
p[i].burst_time);
        p[i].completion_time = current_time + p[i].burst_time;
        p[i].turnaround_time = p[i].completion_time - p[i].arrival_time;
        p[i].waiting_time = p[i].turnaround_time - p[i].burst_time;
        current_time += p[i].burst_time;
    }
    printf("Finish\n");
}

void calculate_and_display_avg_times(struct Process p[], int n) {
    float total_turnaround_time = 0, total_waiting_time = 0;
    printf("\nProcess\tArrival\tBurst\tCompletion\tTAT\tWT\n");
    for (int i = 0; i < n; i++) {
        total_turnaround_time += p[i].turnaround_time;
        total_waiting_time += p[i].waiting_time;
        printf("P%d\t%d\t%d\t%d\t\t%d\t%d\n", p[i].pid, p[i].arrival_time,
p[i].burst_time,
               p[i].completion_time, p[i].turnaround_time,
p[i].waiting_time);
    }
    printf("\nAverage Turnaround Time: %.2f", total_turnaround_time / n);
    printf("\nAverage Waiting Time: %.2f\n", total_waiting_time / n);
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
```

```
    struct Process p[n];

    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("Enter arrival time for process P%d: ", i + 1);
        scanf("%d", &p[i].arrival_time);
        printf("Enter burst time for process P%d: ", i + 1);
        scanf("%d", &p[i].burst_time);
    }

    sort_by_arrival(p, n);
    fcfs_scheduling(p, n);
    calculate_and_display_avg_times(p, n);

    return 0;
}
```

Slip no-19

Q.1 Write a C program to implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following ‘list’ commands as
myshell$ list f dirname :- To print names of all the files in current directory.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <dirent.h>

#define MAX_CMD_LEN 1024
#define MAX_ARGS 100

void list_files_in_directory(const char *dirname) {
    struct dirent *de;
```

```
    DIR *dr = opendir(dirname);

    if (dr == NULL) {
        printf("Could not open current directory\n");
        return;
    }

    printf("Files in directory %s:\n", dirname);
    while ((de = readdir(dr)) != NULL) {
        printf("%s\n", de->d_name);
    }
    closedir(dr);
}

void execute_command(char **args) {
    pid_t pid = fork();

    if (pid < 0) {
        printf("Failed to create a new process\n");
    } else if (pid == 0) { // Child process
        if (execvp(args[0], args) == -1) {
            perror("myshell");
        }
        exit(EXIT_FAILURE);
    } else { // Parent process
        wait(NULL);
    }
}

int main() {
    char command[MAX_CMD_LEN];
    char *args[MAX_ARGS];
    char *token;
    int should_run = 1;

    while (should_run) {
        printf("myshell$ ");
        fgets(command, MAX_CMD_LEN, stdin);
        command[strlen(command) - 1] = '\0'; // Remove newline character
```

```
        int arg_count = 0;
        token = strtok(command, " ");
        while (token != NULL) {
            args[arg_count++] = token;
            token = strtok(NULL, " ");
        }
        args[arg_count] = NULL;

        if (arg_count == 0) {
            continue; // No input, prompt again
        }

        if (strcmp(args[0], "exit") == 0) {
            should_run = 0;
        } else if (strcmp(args[0], "list") == 0 && arg_count == 3 &&
strcmp(args[1], "f") == 0) {
            list_files_in_directory(args[2]);
        } else {
            execute_command(args);
        }
    }

    return 0;
}
```

Q.2 Write the simulation program for Round Robin scheduling for given time quantum. The arrival time and first CPU-burst of different jobs should be input to the system. Accept no. of Processes, arrival time and burst time. The output should give the Gantt chart, turnaround time and waiting time for each process. Also display the average turnaround time and average waiting time.

```
#include <stdio.h>

struct Process {
    int pid;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int completion_time;
```

```
    int turnaround_time;
    int waiting_time;
};

void round_robin(struct Process p[], int n, int time_quantum) {
    int current_time = 0, completed = 0, i = 0;
    int wait_time = 0, tat = 0;

    printf("\nGantt Chart:\n");

    while (completed != n) {
        if (p[i].remaining_time > 0 && p[i].arrival_time <= current_time)
{
            int time_spent = (p[i].remaining_time > time_quantum) ?
time_quantum : p[i].remaining_time;
            printf("P%d [%d - %d] -> ", p[i].pid, current_time,
current_time + time_spent);

            p[i].remaining_time -= time_spent;
            current_time += time_spent;

            if (p[i].remaining_time == 0) {
                p[i].completion_time = current_time;
                p[i].turnaround_time = p[i].completion_time -
p[i].arrival_time;
                p[i].waiting_time = p[i].turnaround_time -
p[i].burst_time;
                wait_time += p[i].waiting_time;
                tat += p[i].turnaround_time;
                completed++;
            }
        }
        i = (i + 1) % n;
    }

    printf("Finish\n");

    printf("\nProcess\tArrival\tBurst\tCompletion\tTAT\tWT\n");
    for (int i = 0; i < n; i++) {
```

```
        printf("P%d\t%d\t%d\t%d\t\t%d\t%d\n", p[i].pid, p[i].arrival_time,
p[i].burst_time,
               p[i].completion_time, p[i].turnaround_time,
p[i].waiting_time);
    }

    printf("\nAverage Turnaround Time: %.2f", (float)tat / n);
    printf("\nAverage Waiting Time: %.2f\n", (float)wait_time / n);
}

int main() {
    int n, time_quantum;

    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct Process p[n];

    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("Enter arrival time for process P%d: ", i + 1);
        scanf("%d", &p[i].arrival_time);
        printf("Enter burst time for process P%d: ", i + 1);
        scanf("%d", &p[i].burst_time);
        p[i].remaining_time = p[i].burst_time; // Initialize remaining
time
    }

    printf("Enter the time quantum: ");
    scanf("%d", &time_quantum);

    round_robin(p, n, time_quantum);

    return 0;
}
```

Slip no-20

Q.1 Write a C program to implement the shell which displays the command prompt “myshell$”. It accepts the command, tokenize the command line and execute it by creating the child process. Also implement the additional command ‘typeline’ as
typeline -a filename :- To print all lines in the file.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

#define MAX_CMD_LEN 1024
#define MAX_ARGS 100

// Function to implement the 'typeline -a filename' command
void typeline_all(char *filename) {
    FILE *file = fopen(filename, "r");
    char line[256];

    if (file == NULL) {
        perror("myshell");
        return;
    }

    // Print each line of the file
    while (fgets(line, sizeof(line), file)) {
        printf("%s", line);
    }
    fclose(file);
}

// Function to execute regular shell commands
void execute_command(char **args) {
    pid_t pid = fork();

    if (pid < 0) {
        perror("myshell");
    } else if (pid == 0) { // Child process
        if (execvp(args[0], args) == -1) {
```

```
            perror("myshell");
        }
        exit(EXIT_FAILURE);
    } else { // Parent process
        wait(NULL);
    }
}

int main() {
    char command[MAX_CMD_LEN];
    char *args[MAX_ARGS];
    char *token;
    int should_run = 1;

    while (should_run) {
        printf("myshell$ ");
        fgets(command, MAX_CMD_LEN, stdin);
        command[strlen(command) - 1] = '\0'; // Remove newline character

        int arg_count = 0;
        token = strtok(command, " ");
        while (token != NULL) {
            args[arg_count++] = token;
            token = strtok(NULL, " ");
        }
        args[arg_count] = NULL;

        if (arg_count == 0) {
            continue; // No input, prompt again
        }

        // Check for exit command
        if (strcmp(args[0], "exit") == 0) {
            should_run = 0;
        }
        // Implement 'typeline -a filename'
        else if (strcmp(args[0], "typeline") == 0 && arg_count == 3 &&
strcmp(args[1], "-a") == 0) {
            typeline_all(args[2]);
        }
```

```
        // Execute other commands
        else {
            execute_command(args);
        }
    }

    return 0;
}
```