FCFS

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct {
char pname[20];
int at, bt, ct, tat, wt;
int bt1;
} Process;
typedef struct {
int start;
char pname[20];
int end;
} Gantt;
Process processes[10];
Gantt ganttChart[100];
int processCount, ganttIndex = 0;
void addToGanttChart(int start, int end, char pname[]) {
ganttChart[ganttIndex].start = start;
ganttChart[ganttIndex].end = end;
strcpy(ganttChart[ganttIndex].pname, pname);
ganttIndex++;
}
void printGanttChart() {
printf("\nGantt Chart:\n");
for (int i = 0; i < ganttIndex; i++) {
printf("| %s ", ganttChart[i].pname);
}
printf("|\n");
for (int i = 0; i < ganttIndex; i++) {
printf("%d ", ganttChart[i].start);
}
printf("%d\n", ganttChart[ganttIndex - 1].end);
}
void acceptProcessInfo() {
printf("Enter the number of processes: ");
scanf("%d", &processCount);
for (int i = 0; i < processCount; i++) {
printf("Enter process name: ");
scanf("%s", processes[i].pname);
printf("Enter arrival time: ");
scanf("%d", &processes[i].at);
```

```
printf("Enter burst time: ");
scanf("%d", &processes[i].bt);
processes[i].bt1 = processes[i].bt;
}
}
void sortProcessesByArrival() {
for (int i = 0; i < processCount - 1; i++) {
for (int j = i + 1; j < processCount; j++) {
if (processes[i].at > processes[j].at) {
Process temp = processes[i];
processes[i] = processes[j];
processes[j] = temp;
}
}
}
}
void fcfsScheduling() {
int time = 0;
for (int i = 0; i < processCount; i++) {
if (time < processes[i].at) {
time = processes[i].at; // If CPU is idle before the process arrives
}
processes[i].ct = time + processes[i].bt; // Completion time
addToGanttChart(time, processes[i].ct, processes[i].pname);
time = processes[i].ct;
}
}
void calculateTATandWT() {
for (int i = 0; i < processCount; i++) {
processes[i].tat = processes[i].ct - processes[i].at;
processes[i].wt = processes[i].tat - processes[i].bt;
}
}
void displayResults() {
printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\n");
for (int i = 0; i < processCount; i++) {
printf("%s\t%d\t%d\t%d\t%d\t%d\n",
processes[i].pname, processes[i].at, processes[i].bt,
processes[i].ct, processes[i].tat, processes[i].wt);
}
}
void calculateAndPrintAverages() {
float totalTAT = 0, totalWT = 0;
```

```
for (int i = 0; i < processCount; i++) {
totalTAT += processes[i].tat;
totalWT += processes[i].wt;
}
printf("Average Turnaround Time: %.2f\n", totalTAT / processCount);
printf("Average Waiting Time: %.2f\n", totalWT / processCount);
}
int main() {
acceptProcessInfo();
sortProcessesByArrival();
fcfsScheduling();
calculateTATandWT();
displayResults();
printGanttChart(); // Print Gantt chart
calculateAndPrintAverages();
return 0;
}
```

SJF

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct {
char pname[20];
int at, bt, ct, tat, wt;
int bt1;
} Process;
typedef struct {
int start;
char pname[20];
int end;
} Gantt;
Process processes[10];
Gantt ganttChart[100];
int processCount, ganttIndex = 0;
void addToGanttChart(int start, int end, char pname[]) {
ganttChart[ganttIndex].start = start;
ganttChart[ganttIndex].end = end;
strcpy(ganttChart[ganttIndex].pname, pname);
ganttIndex++;
}
void printGanttChart() {
printf("\nGantt Chart:\n");
for (int i = 0; i < ganttIndex; i++) {
printf("| %s ", ganttChart[i].pname);
}
printf("|\n");
for (int i = 0; i < ganttIndex; i++) {
printf("%d ", ganttChart[i].start);
}
printf("%d\n", ganttChart[ganttIndex - 1].end);
}
void acceptProcessInfo() {
printf("Enter the number of processes: ");
scanf("%d", &processCount);
for (int i = 0; i < processCount; i++) {
printf("Enter process name: ");
scanf("%s", processes[i].pname);
printf("Enter arrival time: ");
scanf("%d", &processes[i].at);
```

```
printf("Enter burst time: ");
scanf("%d", &processes[i].bt);
processes[i].bt1 = processes[i].bt;
}
}
void sortProcessesByArrival() {
for (int i = 0; i < processCount - 1; i++) {
for (int j = i + 1; j < processCount; j++) {
if (processes[i].at > processes[j].at) {
Process temp = processes[i];
processes[i] = processes[j];
processes[j] = temp;
}
}
}
}
int getShortestJob(int time) {
int minIndex = -1;
int minBurstTime = 9999; // Initialize to a large number
for (int i = 0; i < processCount; i++) {
if (processes[i].at <= time && processes[i].bt1 > 0) {
if (processes[i].bt1 < minBurstTime) {
minBurstTime = processes[i].bt1;
minIndex = i;
}
}
}
return minIndex;
}
void sjfScheduling() {
int time = 0;
int completed = 0;
while (completed < processCount) {
int sjIndex = getShortestJob(time);
if (sjIndex == -1) {
time++; // CPU is idle
} else {
addToGanttChart(time, time + processes[sjIndex].bt1,
processes[sjIndex].pname);
processes[sjIndex].ct = time + processes[sjIndex].bt1;
time = processes[sjIndex].ct;
processes[sjIndex].bt1 = 0; // Process completed
completed++;
```

```
}
}
}
void calculateTATandWT() {
for (int i = 0; i < processCount; i++) {
processes[i].tat = processes[i].ct - processes[i].at;
processes[i].wt = processes[i].tat - processes[i].bt;
}
}
void displayResults() {
printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\n");
for (int i = 0; i < processCount; i++) {
printf("%s\t%d\t%d\t%d\t%d\t%d\n",
processes[i].pname, processes[i].at, processes[i].bt,
processes[i].ct, processes[i].tat, processes[i].wt);
}
}
void calculateAndPrintAverages() {
float totalTAT = 0, totalWT = 0;
for (int i = 0; i < processCount; i++) {
totalTAT += processes[i].tat;
totalWT += processes[i].wt;
}
printf("Average Turnaround Time: %.2f\n", totalTAT / processCount);
printf("Average Waiting Time: %.2f\n", totalWT / processCount);
}
int main() {
acceptProcessInfo();
sortProcessesByArrival();
sjfScheduling();
calculateTATandWT();
displayResults();
printGanttChart(); // Print Gantt chart
calculateAndPrintAverages();
return 0;
}
```

Priority Scheduling

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct {
 char pname[20];
 int at, bt, ct, tat, wt;
 int bt1, priority;
} Process;
typedef struct {
 int start;
 char pname[20];
 int end;
} Gantt;
Process processes[10];
Gantt ganttChart[100];
int processCount, ganttIndex = 0;
void addToGanttChart(int start, int end, char pname[]) {
 ganttChart[ganttIndex].start = start;
 ganttChart[ganttIndex].end = end;
 strcpy(ganttChart[ganttIndex].pname, pname);
 ganttIndex++;
}
void printGanttChart() {
 printf("\nGantt Chart:\n");
 for (int i = 0; i < ganttIndex; i++) {
 printf("| %s ", ganttChart[i].pname);
 }
 printf("|\n");
 for (int i = 0; i < ganttIndex; i++) {
 printf("%d ", ganttChart[i].start);
 }
 printf("%d\n", ganttChart[ganttIndex - 1].end);
}
void acceptProcessInfo() {
 printf("Enter the number of processes: ");
 scanf("%d", &processCount);
 for (int i = 0; i < processCount; i++) {
 printf("Enter process name: ");
 scanf("%s", processes[i].pname);
 printf("Enter arrival time: ");
 scanf("%d", &processes[i].at);
```

```
 printf("Enter burst time: ");
 scanf("%d", &processes[i].bt);
 printf("Enter priority: ");
 scanf("%d", &processes[i].priority);
 processes[i].bt1 = processes[i].bt;
 }
}
void priorityScheduling() {
 int time = 0, completed = 0;
 while (completed < processCount) {
 int highestPriority = 999;
 int processIndex = -1;
 // Find the process with the highest priority that has arrived
 for (int i = 0; i < processCount; i++) {
 if (processes[i].at <= time && processes[i].bt1 > 0) {
 if (processes[i].priority < highestPriority) {
 highestPriority = processes[i].priority;
 processIndex = i;
 }
 }
 }
 if (processIndex != -1) {
 addToGanttChart(time, time + processes[processIndex].bt1,
processes[processIndex].pname);
 time += processes[processIndex].bt1;
 processes[processIndex].ct = time;
 processes[processIndex].bt1 = 0;
 completed++;
 } else {
 time++; // If no process is available, CPU is idle
 }
 }
}
void calculateTATandWT() {
 for (int i = 0; i < processCount; i++) {
 processes[i].tat = processes[i].ct - processes[i].at;
 processes[i].wt = processes[i].tat - processes[i].bt;
 }
}
void displayResults() {
 printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\n");
 for (int i = 0; i < processCount; i++) {
 printf("%s\t%d\t%d\t%d\t%d\t%d\n",
```

```
 processes[i].pname, processes[i].at, processes[i].bt,
 processes[i].ct, processes[i].tat, processes[i].wt);
 }
}
void calculateAndPrintAverages() {
 float totalTAT = 0, totalWT = 0;
 for (int i = 0; i < processCount; i++) {
 totalTAT += processes[i].tat;
 totalWT += processes[i].wt;
 }
 printf("Average Turnaround Time: %.2f\n", totalTAT / processCount);
 printf("Average Waiting Time: %.2f\n", totalWT / processCount);
}
int main() {
 acceptProcessInfo();
 priorityScheduling();
 calculateTATandWT();
 displayResults();
 printGanttChart(); // Print Gantt chart
 calculateAndPrintAverages();
 return 0;
}
```

Round Robbin

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct {
 char pname[20];
 int at, bt, ct, tat, wt;
 int bt1;
} Process;
typedef struct {
 int start;
 char pname[20];
 int end;
} Gantt;
Process processes[10];
Gantt ganttChart[100];
int processCount, timeQuantum, ganttIndex = 0;
void addToGanttChart(int start, int end, char pname[]) {
 ganttChart[ganttIndex].start = start;
 ganttChart[ganttIndex].end = end;
 strcpy(ganttChart[ganttIndex].pname, pname);
 ganttIndex++;
}
void printGanttChart() {
 printf("\nGantt Chart:\n");
 for (int i = 0; i < ganttIndex; i++) {
 printf("| %s ", ganttChart[i].pname);
 }
 printf("|\n");
 for (int i = 0; i < ganttIndex; i++) {
 printf("%d ", ganttChart[i].start);
 }
 printf("%d\n", ganttChart[ganttIndex - 1].end);
}
void acceptProcessInfo() {
 printf("Enter the number of processes: ");
 scanf("%d", &processCount);
 for (int i = 0; i < processCount; i++) {
 printf("Enter process name: ");
 scanf("%s", processes[i].pname);
 printf("Enter arrival time: ");
 scanf("%d", &processes[i].at);
```

```
 printf("Enter burst time: ");
 scanf("%d", &processes[i].bt);
 processes[i].bt1 = processes[i].bt;
 }
 printf("Enter time quantum: ");
 scanf("%d", &timeQuantum);
}
void roundRobinScheduling() {
 int time = 0, completed = 0;
 while (completed < processCount) {
 for (int i = 0; i < processCount; i++) {
 if (processes[i].bt1 > 0 && processes[i].at <= time) {
 int execTime = (processes[i].bt1 > timeQuantum) ? timeQuantum :
Processes[i].bt1;

 addToGanttChart(time, time + execTime, processes[i].pname);
 time += execTime;
 processes[i].bt1 -= execTime;
 if (processes[i].bt1 == 0) {
 processes[i].ct = time;
 completed++;
 }
 }
 }
 }
}
void calculateTATandWT() {
 for (int i = 0; i < processCount; i++) {
 processes[i].tat = processes[i].ct - processes[i].at;
 processes[i].wt = processes[i].tat - processes[i].bt;
 }
}
void displayResults() {
 printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\n");
 for (int i = 0; i < processCount; i++) {
 printf("%s\t%d\t%d\t%d\t%d\t%d\n",
 processes[i].pname, processes[i].at, processes[i].bt,
 processes[i].ct, processes[i].tat, processes[i].wt);
 }
}
void calculateAndPrintAverages() {
 float totalTAT = 0, totalWT = 0;
 for (int i = 0; i < processCount; i++) {
```

```
 totalTAT += processes[i].tat;
 totalWT += processes[i].wt;
 }
 printf("Average Turnaround Time: %.2f\n", totalTAT / processCount);
 printf("Average Waiting Time: %.2f\n", totalWT / processCount);
}
int main() {
 acceptProcessInfo();
 roundRobinScheduling();
 calculateTATandWT();
 displayResults();
 printGanttChart(); // Print Gantt chart
 calculateAndPrintAverages();
 return 0;
}
```

List myshell

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <dirent.h>
// Constants for command input
#define MAX_INPUT 80
#define MAX_ARGS 10
// Function to tokenize command input
int make_toks(char *input, char *args[]) {
 int i = 0;
 char *token;
 token = strtok(input, " ");
 while (token != NULL) {
 args[i++] = token;
 token = strtok(NULL, " ");
 }
 args[i] = NULL; // Null-terminate the arguments array
 return i; // Return number of tokens
}
// Function to handle the 'list' command
void list(char *dirname, char option) {
 DIR *dp = opendir(dirname);
```

```
 if (dp == NULL) {
 printf("Directory %s not found.\n", dirname);
 return;
 }
 struct dirent *entry;
 if (option == 'f') { // List files
 while ((entry = readdir(dp)) != NULL) {
 if (entry->d_type == DT_REG) // Regular file
 printf("%s\n", entry->d_name);
 }
 } else if (option == 'n') { // Count files
 int fileCount = 0;
 while ((entry = readdir(dp)) != NULL) {
 if (entry->d_type == DT_REG) fileCount++;
 }
 printf("Total files: %d\n", fileCount);
 }
 closedir(dp);
}
// Main shell loop
void myshell() {
 char input[MAX_INPUT];
 char *args[MAX_ARGS];
 while (1) {
 printf("myshell$ ");
 fflush(stdout);
 fgets(input, sizeof(input), stdin);
 // Remove trailing newline character
 input[strcspn(input, "\n")] = 0;
 // Tokenize the input
 int n = make_toks(input, args);
 // Handle built-in commands
 if (n > 0) {
 if (strcmp(args[0], "exit") == 0) {
 exit(0); // Exit the shell
 } else if (strcmp(args[0], "list") == 0 && n == 3) {
 list(args[1], args[2][0]); // Call list function
 } else {
 printf("Invalid command.\n");
 }
 }
 }
}
```

```
int main() {
 myshell(); // Start the shell
 return 0;
}
```

Fifo

```
#include <stdio.h>
void display(int frames[], int n) {
 for (int i = 0; i < n; i++) {
 printf(frames[i] == -1 ? " - " : " %d ", frames[i]);
 }
 printf("\n");
}
int isPageInFrame(int page, int frames[], int n) {
 for (int i = 0; i < n; i++) {
 if (frames[i] == page) return 1;
 }
 return 0;
}
void runFIFO(int ref_string[], int ref_len, int n) {
 int frames[n], page_faults = 0, next = 0;
 for (int i = 0; i < n; i++) frames[i] = -1;
 printf("Page Replacement Process (FIFO):\n");
 for (int i = 0; i < ref_len; i++) {
 int page = ref_string[i];
 if (!isPageInFrame(page, frames, n)) {
 frames[next] = page;
 next = (next + 1) % n;
 page_faults++;
 }
 display(frames, n);
 }
 printf("Total Page Faults: %d\n\n", page_faults);
}
int main() {
 int n;
 printf("Enter number of frames: ");
 scanf("%d", &n);
 int ref_string[] = {3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6};
```

```
 int ref_len = sizeof(ref_string) / sizeof(ref_string[0]);
 runFIFO(ref_string, ref_len, n);
 return 0;
}
```

Lifo

```
#include <stdio.h>
#include<stdlib.h>
void display(int frames[], int n) {
 for (int i = 0; i < n; i++) {
 printf(frames[i] == -1 ? " - " : " %d ", frames[i]);
 }
 printf("\n");
}
void displayFaultFrames(int fault_frames[], int fault_count) {
 printf("Fault Frames: ");
 for (int i = 0; i < fault_count; i++) {
 printf("%d ", fault_frames[i]);
 }
 printf("\n");
}
int isPageInFrame(int page, int frames[], int n) {
 for (int i = 0; i < n; i++) {
 if (frames[i] == page) return 1;
 }
 return 0;
}
int findFreeFrame(int frames[], int n) {
 for (int i = 0; i < n; i++) {
 if (frames[i] == -1) return i;
 }
 return -1;
}
int findLFU(int freq[], int n) {
 int min = freq[0], index = 0;
 for (int i = 1; i < n; i++) {
 if (freq[i] < min) {
 min = freq[i];
 index = i;
 }
```

```
 }
 return index;
}
void runLFU(int ref_string[], int ref_len, int n) {
 int frames[n], freq[n], page_faults = 0;
 int fault_frames[n]; // To store pages causing faults
 int fault_count = 0;
 for (int i = 0; i < n; i++) {
 frames[i] = -1;
 freq[i] = 0;
 }
 printf("Page Replacement Process (LFU):\n");
 for (int i = 0; i < ref_len; i++) {
 int page = ref_string[i];
 if (!isPageInFrame(page, frames, n)) {
 int free_frame = findFreeFrame(frames, n);
 if (free_frame == -1) {
 int lfu_index = findLFU(freq, n);
 frames[lfu_index] = page;
 } else {
 frames[free_frame] = page;
 }
 fault_frames[fault_count++] = page; // Store fault page
 page_faults++;
 } else {
 for (int j = 0; j < n; j++) {
 if (frames[j] == page) freq[j]++;
 }
 }
 display(frames, n);
 }
 printf("Total Page Faults: %d\n", page_faults);
 displayFaultFrames(fault_frames, fault_count); // Display fault frames
}
int main() {
 int n;
 printf("Enter number of frames: ");
 scanf("%d", &n);
 int ref_string[] = {3, 4, 5, 4, 3, 4, 7, 2, 4, 5, 6, 7, 2, 4, 6};
 int ref_len = sizeof(ref_string) / sizeof(ref_string[0]);
 runLFU(ref_string, ref_len, n);
 return 0;
```

Lru

```c
#include <stdio.h>
void display(int frames[], int n) {
 for (int i = 0; i < n; i++) {
 printf(frames[i] == -1 ? " - " : " %d ", frames[i]);
 }
 printf("\n");
}
void displayFaultFrames(int fault_frames[], int fault_count) {
 printf("Fault Frames: ");
 for (int i = 0; i < fault_count; i++) {
 printf("%d ", fault_frames[i]);
 }
 printf("\n");
}
int isPageInFrame(int page, int frames[], int n) {
 for (int i = 0; i < n; i++) {
 if (frames[i] == page) return i;
 }
 return -1;
}
int findLRU(int time[], int n) {
 int min_time = time[0], index = 0;
 for (int i = 1; i < n; i++) {
 if (time[i] < min_time) {
 min_time = time[i];
 index = i;
 }
 }
 return index;
}
void runLRU(int ref_string[], int ref_len, int n) {
 int frames[n], page_faults = 0, time[n];
 int fault_frames[n]; // To store pages causing faults
 int fault_count = 0;
 for (int i = 0; i < n; i++) {
 frames[i] = -1;
 time[i] = 0;
 }
```

```
 printf("Page Replacement Process (LRU):\n");
 for (int i = 0; i < ref_len; i++) {
 int page = ref_string[i];
 int index = isPageInFrame(page, frames, n);
 if (index == -1) {
 int lru_index = findLRU(time, n);
 frames[lru_index] = page;
 fault_frames[fault_count++] = page; // Store fault page
 page_faults++;
 }
 for (int j = 0; j < n; j++) {
 if (frames[j] == page) time[j] = i + 1;
 }
 display(frames, n);
 }
 printf("Total Page Faults: %d\n", page_faults);
 displayFaultFrames(fault_frames, fault_count); // Display fault frames
}
int main() {
 int n;
 printf("Enter number of frames: ");
 scanf("%d", &n);
 int ref_string[] = {3, 5, 7, 2, 5, 1, 2, 3, 1, 3, 5, 3, 1, 6, 2};
 int ref_len = sizeof(ref_string) / sizeof(ref_string[0]);
 runLRU(ref_string, ref_len, n);
 return 0;
}
```

Mfu

```c
#include <stdio.h>
void display(int frames[], int n) {
 for (int i = 0; i < n; i++) {
 printf(frames[i] == -1 ? " - " : " %d ", frames[i]);
 }
 printf("\n");
}
void displayFaultFrames(int fault_frames[], int fault_count) {
 printf("Fault Frames: ");
 for (int i = 0; i < fault_count; i++) {
 printf("%d ", fault_frames[i]);
 }
 printf("\n");
}
int isPageInFrame(int page, int frames[], int n) {
 for (int i = 0; i < n; i++) {
 if (frames[i] == page) return 1;
 }
 return 0;
}
int findMFU(int freq[], int n) {
 int max = freq[0], index = 0;
 for (int i = 1; i < n; i++) {
 if (freq[i] > max) {
 max = freq[i];
 index = i;
 }
 }
 return index;
}
void runMFU(int ref_string[], int ref_len, int n) {
 int frames[n], freq[n], page_faults = 0;
 int fault_frames[n]; // To store pages causing faults
 int fault_count = 0;
 for (int i = 0; i < n; i++) {
 frames[i] = -1;
 freq[i] = 0;
 }
 printf("Page Replacement Process (MFU):\n");
 for (int i = 0; i < ref_len; i++) {
 int page = ref_string[i];
```

```
 if (!isPageInFrame(page, frames, n)) {
 int mfu_index = findMFU(freq, n);
 frames[mfu_index] = page;
 fault_frames[fault_count++] = page; // Store fault page
 page_faults++;
 } else {
 for (int j = 0; j < n; j++) {
 if (frames[j] == page) freq[j]++;
 }
 }
 display(frames, n);
 }
 printf("Total Page Faults: %d\n", page_faults);
 displayFaultFrames(fault_frames, fault_count); // Display fault frames
}
int main() {
 int n;
 printf("Enter number of frames: ");
 scanf("%d", &n);
 int ref_string[] = {8, 5, 7, 8, 5, 7, 2, 3, 7, 3, 5, 9, 4, 6, 2};
 int ref_len = sizeof(ref_string) / sizeof(ref_string[0]);
 runMFU(ref_string, ref_len, n);
 return 0;
}
```

Typeline commands

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
// Constants for command input
#define MAX_INPUT 80
#define MAX_ARGS 10
// Function to tokenize command input
int make_toks(char *input, char *args[]) {
 int i = 0;
 char *token;
 token = strtok(input, " ");
 while (token != NULL) {
 args[i++] = token;
 token = strtok(NULL, " ");
 }
 args[i] = NULL; // Null-terminate the arguments array
 return i; // Return number of tokens
}
// Function to handle the 'typeline' command
void typeline(char *option, char *filename) {
 FILE *file = fopen(filename, "r");
 if (!file) {
 printf("File %s not found.\n", filename);
 return;
 }
 if (strcmp(option, "a") == 0) {
 char line[256];
 while (fgets(line, sizeof(line), file)) {
 printf("%s", line);
 }
 } else {
 int n = atoi(option);
 for (int i = 0; i < n && !feof(file); i++) {
 char line[256];
 fgets(line, sizeof(line), file);
 printf("%s", line);
 }
 }
 fclose(file);
}
```

```
// Main shell loop
void myshell() {
 char input[MAX_INPUT];
 char *args[MAX_ARGS];
 while (1) {
 printf("myshell$ ");
 fflush(stdout);
 fgets(input, sizeof(input), stdin);
 // Remove trailing newline character
 input[strcspn(input, "\n")] = 0;
 // Tokenize the input
 int n = make_toks(input, args);
 // Handle built-in commands
 if (n > 0) {
 if (strcmp(args[0], "exit") == 0) {
 exit(0); // Exit the shell
 } else if (strcmp(args[0], "typeline") == 0 && n == 3) {
 typeline(args[1], args[2]); // Call typeline function
 } else {
 printf("Invalid command.\n");
 }
 }
 }
}
int main() {
 myshell(); // Start the shell
 return 0;
}
```

Optimal

```
#include <stdio.h>
void display(int frames[], int n) {
 for (int i = 0; i < n; i++) {
 printf(frames[i] == -1 ? " - " : " %d ", frames[i]);
 }
 printf("\n");
}
int isPageInFrame(int page, int frames[], int n) {
 for (int i = 0; i < n; i++) {
 if (frames[i] == page) return 1;
```

```
 }
 return 0;
}
int findOptimal(int frames[], int n, int ref_string[], int ref_len, int
current_index) {
 int farthest = -1, index = -1;
 for (int i = 0; i < n; i++) {
 int j;
 for (j = current_index; j < ref_len; j++) {
 if (frames[i] == ref_string[j]) {
 if (j > farthest) {
 farthest = j;
 index = i;
 }
 break;
 }
 }
 if (j == ref_len) return i;
 }
 return (index != -1) ? index : 0;
}
void runOptimal(int ref_string[], int ref_len, int n) {
 int frames[n], page_faults = 0;
 for (int i = 0; i < n; i++) frames[i] = -1;
 printf("Page Replacement Process (Optimal):\n");
 for (int i = 0; i < ref_len; i++) {
 int page = ref_string[i];
 if (!isPageInFrame(page, frames, n)) {
 int replace_index = findOptimal(frames, n, ref_string, ref_len, i);
 frames[replace_index] = page;
 page_faults++;
 }
 display(frames, n);
 }
 printf("Total Page Faults: %d\n\n", page_faults);
}
int main() {
 int n;
 printf("Enter number of frames: ");
 scanf("%d", &n);
 int ref_string[] = {3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6};
 int ref_len = sizeof(ref_string) / sizeof(ref_string[0]);
 runOptimal(ref_string, ref_len, n);
```

```
 return 0;
}
```

Count

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
// Constants for command input
#define MAX_INPUT 80
#define MAX_ARGS 10
// Function to tokenize command input
int make_toks(char *input, char *args[]) {
 int i = 0;
 char *token;
 token = strtok(input, " ");
 while (token != NULL) {
 args[i++] = token;
 token = strtok(NULL, " ");
 }
 args[i] = NULL; // Null-terminate the arguments array
 return i; // Return number of tokens
}
// Function to count characters, words, and lines in a file
void count_file(char *option, char *filename) {
 FILE *file = fopen(filename, "r");
 if (!file) {
 printf("File %s not found.\n", filename);
 return;
 }
 char c;
 int charCount = 0, wordCount = 0, lineCount = 0;
 while ((c = fgetc(file)) != EOF) {
 charCount++;
 if (c == ' ' || c == '\n') wordCount++;
 if (c == '\n') lineCount++;
 }
 if (lineCount > 0) lineCount++; // Adjust for the last line if not
terminated by newline
 if (strcmp(option, "c") == 0) {
```

```
 printf("Number of characters: %d\n", charCount);
 } else if (strcmp(option, "w") == 0) {
 printf("Number of words: %d\n", wordCount);
 } else if (strcmp(option, "l") == 0) {
 printf("Number of lines: %d\n", lineCount);
 }
 fclose(file);
}
// Main shell loop
void myshell() {
 char input[MAX_INPUT];
 char *args[MAX_ARGS];
 while (1) {
 printf("myshell$ ");
 fflush(stdout);
 fgets(input, sizeof(input), stdin);
 // Remove trailing newline character
 input[strcspn(input, "\n")] = 0;
 // Tokenize the input
 int n = make_toks(input, args);
 // Handle built-in commands
 if (n > 0) {
 if (strcmp(args[0], "exit") == 0) {
 exit(0); // Exit the shell
 } else if (strcmp(args[0], "count") == 0 && n == 3) {
 count_file(args[1], args[2]); // Call count function
 } else {
 printf("Invalid command.\n");
 }
 }
 }
}
int main() {
 myshell(); // Start the shell
 return 0;
}
```