PROGRAMME-6

AIM: CPU Scheduling Algorithms in C:

a)First Come first serve(FCFS)

b)Shortest job first(SJF)

c)Shortest Remaining Time Next (SRTN)

d)Round Robin

e)Priority Source

code:

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

void fcfs(int n, int bt[], int at[]) {     int
wt[n], tat[n], total_wt = 0, total_tat = 0;
wt[0] = 0;

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

   printf("\nFCFS Scheduling:\n");
   printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\n");
for (int i = 0; i < n; i++) {         tat[i] = bt[i] + wt[i];         total_wt +=
wt[i];         total_tat += tat[i];
      printf("%d\t%d\t\t%d\t\t%d\n", i + 1, bt[i], wt[i], tat[i]);
   }
   printf("Average Waiting Time: %.2f\n", (float)total_wt / n);
printf("Average Turnaround Time: %.2f\n", (float)total_tat / n);
}

void sjf(int n, int bt[]) {
   int wt[n], tat[n], total_wt = 0, total_tat = 0, completed[n], time = 0, min_bt, shortest;
for (int i = 0; i < n; i++) completed[i] = 0;
```

```
   printf("\nSJF Scheduling:\n");
   printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\n");
for (int count = 0; count < n; count++) {         min_bt = INT_MAX;
       shortest = -1;         for
(int i = 0; i < n; i++) {
           if (!completed[i] && bt[i] < min_bt) {
min_bt = bt[i];                 shortest = i;
           }
       }
       time += bt[shortest];
wt[shortest] = time - bt[shortest];
tat[shortest] = time;
completed[shortest] = 1;
total_wt += wt[shortest];
total_tat += tat[shortest];
       printf("%d\t%d\t\t%d\t\t%d\n", shortest + 1, bt[shortest], wt[shortest], tat[shortest]);
   }
   printf("Average Waiting Time: %.2f\n", (float)total_wt / n);
printf("Average Turnaround Time: %.2f\n", (float)total_tat / n);
}

void rr(int n, int bt[], int quantum) {
   int wt[n], tat[n], rem_bt[n], total_wt = 0, total_tat = 0, time =
0;     for (int i = 0; i < n; i++) rem_bt[i] = bt[i];

   printf("\nRound Robin Scheduling:\n");
   printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\n");
while (1) {         int done = 1;         for (int i = 0; i < n; i++) {
if (rem_bt[i] > 0) {                 done = 0;
           if (rem_bt[i] > quantum) {
time += quantum;
rem_bt[i] -= quantum;
           } else {
```

```
                time += rem_bt[i];
wt[i] = time - bt[i];
rem_bt[i] = 0;
            }
        }
    }
    if (done) break;
  }

  for (int i = 0; i < n; i++) {
tat[i] = bt[i] + wt[i];
total_wt += wt[i];
total_tat += tat[i];
    printf("%d\t%d\t\t%d\t\t%d\n", i + 1, bt[i], wt[i], tat[i]);
  }
  printf("Average Waiting Time: %.2f\n", (float)total_wt / n);
printf("Average Turnaround Time: %.2f\n", (float)total_tat / n);
}

void priority(int n, int bt[], int pr[]) {
  int wt[n], tat[n], total_wt = 0, total_tat = 0, completed[n], time = 0, max_pr, highest;
for (int i = 0; i < n; i++) completed[i] = 0;

  printf("\nPriority Scheduling:\n");
  printf("Process\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n");
for (int count = 0; count < n; count++) {      max_pr = INT_MIN;      highest
= -1;       for (int i = 0; i < n; i++) {
        if (!completed[i] && pr[i] > max_pr) {
max_pr = pr[i];              highest = i;
        }
    }
    time += bt[highest];
wt[highest] = time - bt[highest];
tat[highest] = time;
```

```
completed[highest] = 1;
total_wt += wt[highest];
total_tat += tat[highest];
      printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", highest + 1, bt[highest], pr[highest], wt[highest],
tat[highest]);
   }
   printf("Average Waiting Time: %.2f\n", (float)total_wt / n);
printf("Average Turnaround Time: %.2f\n", (float)total_tat / n);
}

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

   int bt[n], at[n], pr[n];
   printf("Enter burst times for each process:\n");
for (int i = 0; i < n; i++) {         printf("Process
%d: ", i + 1);         scanf("%d", &bt[i]);
   }

   printf("Enter arrival times for each process:\n");
for (int i = 0; i < n; i++) {         printf("Process
%d: ", i + 1);         scanf("%d", &at[i]);
   }

   printf("Enter priorities for each process:\n");
for (int i = 0; i < n; i++) {         printf("Process
%d: ", i + 1);         scanf("%d", &pr[i]);
   }

   printf("Enter time quantum for Round Robin: ");
scanf("%d", &quantum);

   // Execute all scheduling algorithms
```

```
    fcfs(n, bt, at);
    sjf(n, bt);     rr(n,
bt, quantum);
priority(n, bt, pr);

    return 0;
}
```

Output:

```
PS D:\OneDrive\Desktop\os lab> cd "d:\OneDrive\Desktop\os lab\" ; if ($?) { gcc cpu-scheduling.c -o cpu-scheduling } ; if ($?) { .\cpu-scheduling }
Enter the number of processes: 4
Enter burst times for each process:
Process 1: 23
Process 2: 45
Process 3: 67
Process 4: 89
Enter arrival times for each process:
Process 1: 2
Process 2: 3
Process 3: 5
Process 4: 6
Enter priorities for each process:
Process 1: 4
Process 2: 3
Process 3: 2
Process 4: 0
Enter time quantum for Round Robin: 34

FCFS Scheduling:
Process Burst Time      Waiting Time    Turnaround Time
1       23              0               23
2       45              23              68
3       67              68              135
4       89              135             224
Average Waiting Time: 56.50
Average Turnaround Time: 112.50

SJF Scheduling:
Process Burst Time      Waiting Time    Turnaround Time
1       23              0               23
2       45              23              68
3       67              68              135
4       89              135             224
Average Waiting Time: 56.50
Average Turnaround Time: 112.50

Round Robin Scheduling:
Process Burst Time      Waiting Time    Turnaround Time
1       23              0               23
2       45              91              136
3       67              102             169
4       89              135             224
Average Waiting Time: 82.00
Average Turnaround Time: 138.00

Priority Scheduling:
Process Burst Time      Priority        Waiting Time    Turnaround Time
1       23              4               0               23
2       45              3               23              68
3       67              2               68              135
4       89              0               135             224
Average Waiting Time: 56.50
Average Turnaround Time: 112.50
PS D:\OneDrive\Desktop\os lab>
```

PROGRAMME-7

AIM: WAP to implement page replacement algorithms

a) **FIFO**
b) **LRU**
c) **LFU**
d) **Optimal**

Source code:

```
#include <stdio.h> #include
<limits.h>
void fifo(int pages[], int n, int capacity) {     int
frame[capacity], front = 0, count = 0, faults = 0;
for (int i = 0; i < capacity; i++) frame[i] = -1;     for
(int i = 0; i < n; i++) {         int found = 0;         for
(int j = 0; j < capacity; j++) {             if (frame[j] ==
pages[i]) {                 found = 1;                 break;
            }         }         if (!found) {
frame[front] = pages[i];             front
= (front + 1) % capacity;
faults++;
        }
    }
    printf("FIFO Page Faults: %d\n", faults);
}
void lru(int pages[], int n, int capacity) {     int
frame[capacity], time[capacity], faults = 0, clock = 0;
for (int i = 0; i < capacity; i++) frame[i] = -1;         for (int i
= 0; i < n; i++) {         int found = 0, lru_index = 0;

        for (int j = 0; j < capacity; j++) {
if (frame[j] == pages[i]) {
```

```
found = 1;                time[j] =
clock++;                break;
            }        }        if (!found) {
int min_time = INT_MAX;            for
(int j = 0; j < capacity; j++) {
if (time[j] < min_time) {
min_time = time[j];
lru_index = j;
                }
            }
            frame[lru_index] = pages[i];
time[lru_index] = clock++;
faults++;
        }
    }
    printf("LRU Page Faults: %d\n", faults);
}
void lfu(int pages[], int n, int capacity) {     int
frame[capacity], freq[capacity], faults = 0;
for (int i = 0; i < capacity; i++) {         frame[i]
= -1;         freq[i] = 0;
    }

    for (int i = 0; i < n; i++) {         int
found = 0, lfu_index = 0;         for
(int j = 0; j < capacity; j++) {
if (frame[j] == pages[i]) {
found = 1;                freq[j]++;
break;
            }        }        if (!found) {
int min_freq = INT_MAX;            for
```

```
(int j = 0; j < capacity; j++) {
if (freq[j] < min_freq) {
min_freq = freq[j];
lfu_index = j;
            }
          }
          frame[lfu_index] = pages[i];
freq[lfu_index] = 1;
faults++;
       }
   }
   printf("LFU Page Faults: %d\n", faults);
}
void optimal(int pages[], int n, int capacity) {
int frame[capacity], faults = 0;         for (int i = 0;
i < capacity; i++) frame[i] = -1;          for (int i =
0; i < n; i++) {         int found = 0, replace_index
= 0;         for (int j = 0; j < capacity; j++) {
if (frame[j] == pages[i]) {                found = 1;
break;
          }        }        if
(!found) {             int
farthest = -1;
for (int j = 0; j <
capacity; j++) {
int next_use =
INT_MAX;
for (int k = i + 1; k < n;
k++) {                   if
(frame[j] == pages[k])
{
```

```
next_use = k;
break;
                }
            }               if (next_use
> farthest) {                   farthest
= next_use;
replace_index = j;
            }
        }
        frame[replace_index] = pages[i];
faults++;
    } }
   printf("Optimal Page Faults: %d\n", faults);
} int main() {     int pages[] = {7, 0, 1, 2, 0, 3, 0,
4, 2, 3, 0, 3, 2};     int n = sizeof(pages) /
sizeof(pages[0]);     int capacity = 3;
fifo(pages, n, capacity);     lru(pages, n, capacity);
lfu(pages, n, capacity);     optimal(pages, n,
capacity);     return 0;
}
```

OUTPUT:

PROGRAMME-8

AIM: Write a programme to check if thrashing happens in fifo or not.

Source code:

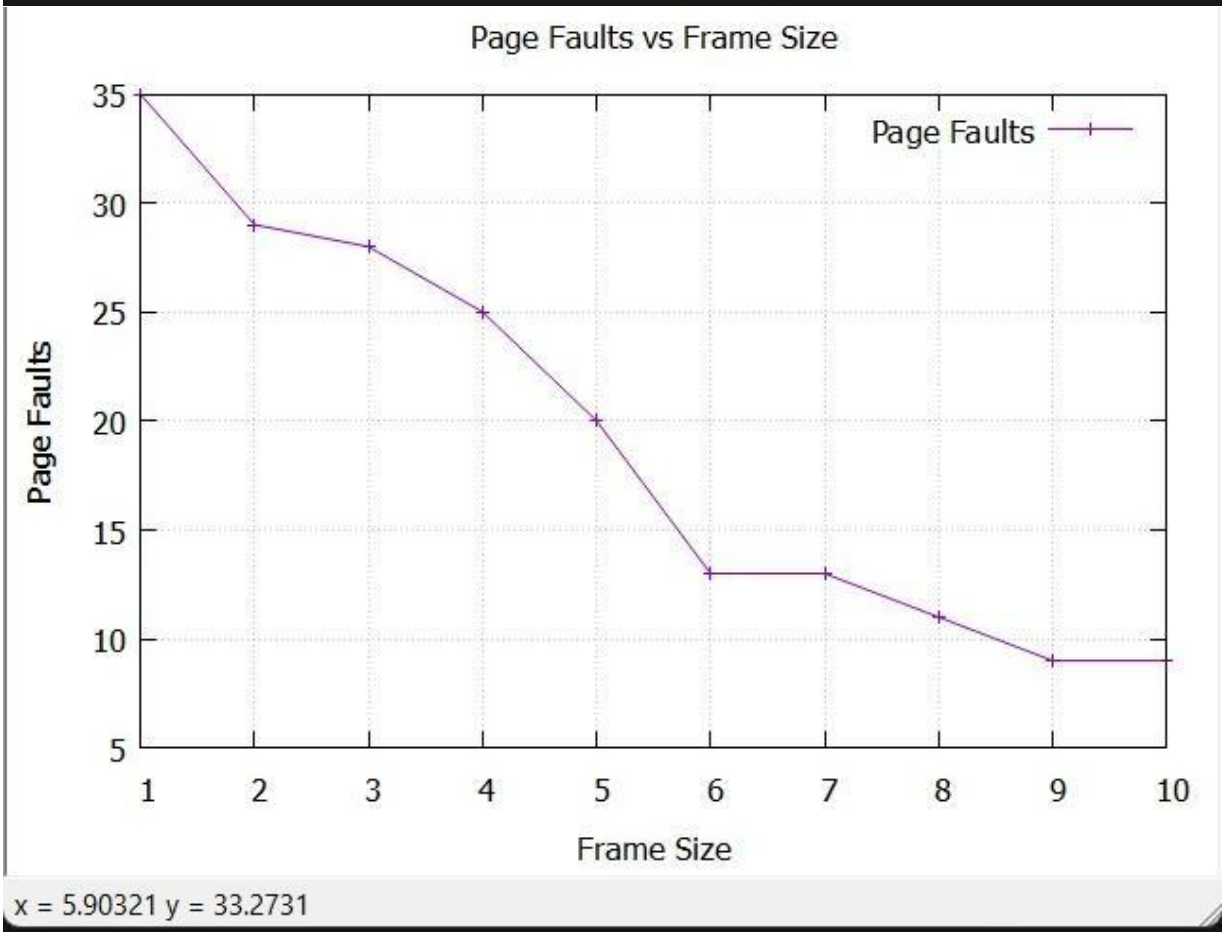
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define TOTAL_REFERENCES 40  // Total page references
#define MAX_FRAMES 10        // Max number of frames to check for thrashing
#define PAGE_RANGE 10        // Pages numbered from 0 to 9
int fifo_page_faults(int pages[], int n, int frames) {
   int queue[MAX_FRAMES], front = 0, rear = 0, page_faults = 0;
int present[MAX_FRAMES] = {0}; // Tracks if a page is in memory
for (int i = 0; i < n; i++) {        int page = pages[i];        int found = 0;

      // Check if page is already in queue
for (int j = 0; j < rear; j++) {            if
(queue[j] == page) {                found =
1;                break;
          }
}
         if (!found) {            if
(rear < frames) {
queue[rear++] = page;
          } else {
             queue[front] = page;
             front = (front + 1) % frames; // FIFO replacement
          }
          page_faults++;
       } }
   return page_faults;
} int main()
{
   int pages[TOTAL_REFERENCES];
   srand(time(NULL));
```

```
   printf("Generated Page References:\n");     for
(int i = 0; i < TOTAL_REFERENCES; i++) {
pages[i] = rand() % PAGE_RANGE;
printf("%d ", pages[i]);
   }
   printf("\n\n");
   int faults[MAX_FRAMES] = {0};
int thrashing[MAX_FRAMES] = {0};
   // Sequential computation of page faults for different frame sizes
for (int frames = 1; frames <= MAX_FRAMES; frames++) {
       faults[frames - 1] = fifo_page_faults(pages, TOTAL_REFERENCES, frames);
   }
   printf("Frames\tPage Faults\tThrashing?\n");     printf("---------
------------------------\n");
   FILE *data_file = fopen("page_faults_data.dat", "w");
if (data_file == NULL) {         perror("Error opening
file");         return 1;
   }
   for (int frames = 1; frames <= MAX_FRAMES; frames++) {
if (frames > 1 && faults[frames - 1] > faults[frames - 2]) {
thrashing[frames - 1] = 1; // Thrashing detected
       }
       printf("%d\t%d\t    %s\n", frames, faults[frames - 1], thrashing[frames - 1] ? "YES" : "NO");
fprintf(data_file, "%d %d\n", frames, faults[frames - 1]); // Write frame size and faults
   }

   fclose(data_file);

   // Generate graph using gnuplot
   FILE *gnuplot_script = popen("gnuplot -persistent", "w");
if (gnuplot_script == NULL) {         perror("Error opening
gnuplot");         return 1;
   }
```

```
    fprintf(gnuplot_script, "set title 'Page Faults vs Frame
Size'\n");
    fprintf(gnuplot_script, "set xlabel 'Frame Size'\n");
fprintf(gnuplot_script, "set ylabel 'Page Faults'\n");
fprintf(gnuplot_script, "set grid\n");
    fprintf(gnuplot_script, "plot 'page_faults_data.dat' using 1:2 with linespoints title 'Page Faults'\n");
pclose(gnuplot_script);     return 0;
}
```

OUTPUT:

```
PS D:\OneDrive\Desktop\os lab> cd "d:\OneDrive\Desktop\os lab\" ; if ($?) { gcc trashing.c -o trashing } ; if ($?) { .\trashing }
Generated Page References:
4 9 5 6 9 1 9 1 8 4 5 9 3 3 7 7 8 4 4 6 4 3 1 9 8 2 3 6 3 9 5 5 9 7 6 6 5 4 9 4

Frames  Page Faults     Thrashing?
--------------------------------
1       35              NO
2       29              NO
3       28              NO
4       25              NO
5       20              NO
6       13              NO
7       13              NO
8       11              NO
9       9               NO
10      9               NO
```



PROGRAMME-8b

AIM: Write a programme to check if thrashing happens in fifo or not using parallelism.

Source code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
#define TOTAL_REFERENCES 40  // Total page references
#define MAX_FRAMES 10        // Max number of frames to check for thrashing
#define PAGE_RANGE 10        // Pages numbered from 0 to 9
int fifo_page_faults(int pages[], int n, int frames) {
   int queue[MAX_FRAMES], front = 0, rear = 0, page_faults = 0;
int present[MAX_FRAMES] = {0}; // Tracks if a page is in memory
for (int i = 0; i < n; i++) {        int page = pages[i];        int found = 0;
      for (int j = 0; j < rear; j++) {
if (queue[j] == page) {
found = 1;                break;
         }        if (!found) {
if (rear < frames) {
queue[rear++] = page;
         } else {
            queue[front] = page;
 front = (front + 1) % frames; // FIFO replacement
         }
         page_faults++;
      }
   }
   return page_faults;
} int main()
{
   int pages[TOTAL_REFERENCES];
```

```
    srand(time(NULL));
    printf("Generated Page References:\n");     for
(int i = 0; i < TOTAL_REFERENCES; i++) {
pages[i] = rand() % PAGE_RANGE;
printf("%d ", pages[i]);
    }
    printf("\n\n");
    int faults[MAX_FRAMES] = {0};
int thrashing[MAX_FRAMES] = {0};
    #pragma omp parallel for
    for (int frames = 1; frames <= MAX_FRAMES; frames++) {
        faults[frames - 1] = fifo_page_faults(pages, TOTAL_REFERENCES, frames);
    }
    FILE *data_file = fopen("page_faults_parallel_data.dat", "w");
if (data_file == NULL) {         perror("Error opening file");
return 1;
    }
    printf("Frames\tPage Faults\tThrashing?\n");     printf("---------
------------------------\n");
    for (int frames = 1; frames <= MAX_FRAMES; frames++) {
if (frames > 1 && faults[frames - 1] > faults[frames - 2]) {
thrashing[frames - 1] = 1; // Thrashing detected
        }
        printf("%d\t%d\t    %s\n", frames, faults[frames - 1], thrashing[frames - 1] ? "YES" : "NO");
fprintf(data_file, "%d %d\n", frames, faults[frames - 1]); // Write frame size and faults
    }
    fclose(data_file);
    FILE *gnuplot_script = popen("gnuplot -persistent", "w");
if (gnuplot_script == NULL) {         perror("Error opening
gnuplot");         return 1;
    }
    fprintf(gnuplot_script, "set title 'Page Faults vs Frame Size (Parallel)'\n");
fprintf(gnuplot_script, "set xlabel 'Frame Size'\n");
```

```
fprintf(gnuplot_script, "set ylabel 'Page Faults'\n");
fprintf(gnuplot_script, "set grid\n");
    fprintf(gnuplot_script, "plot 'page_faults_parallel_data.dat' using 1:2 with linespoints title 'Page Faults'\n");     pclose(gnuplot_script);     return 0;
}
```

OUTPUT:

```
PS D:\OneDrive\Desktop\os lab> cd "d:\OneDrive\Desktop\os lab\" ; if ($?) { gcc thrash-parallel.c -o thrash-parallel } ; if ($?) { .\thrash-parallel }
Generated Page References:
6 0 6 4 4 2 7 8 2 7 8 9 7 6 8 2 6 1 2 1 1 0 5 7 4 5 6 8 2 8 9 5 5 5 8 9 0 8 4 7

Frames  Page Faults     Thrashing?
---------------------------------
1       36              NO
2       33              NO
3       23              NO
4       23              NO
5       22              NO
6       17              NO
7       13              NO
8       12              NO
9       9               NO
10      9               NO
```