FCFS process scheduling alg:

```
#include <stdio.h>

struct Process {
   int pid, AT, BT, CT, TAT, WT;
};
int main() {
   int n;
   printf("Enter number of processes: ");
   scanf("%d", &n);

   struct Process p[n];
   int totalWT = 0, totalTAT = 0;
   // Input AT and BT for each process
   for (int i = 0; i < n; i++) {
      p[i].pid = i + 1;
      printf("Enter AT and BT for process %d: ", p[i].pid);
      scanf("%d %d", &p[i].AT, &p[i].BT);
   }
   // Sort by Arrival Time (AT)
   for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
         if (p[i].AT > p[j].AT) {
            struct Process temp = p[i];
            p[i] = p[j];
            p[j] = temp;
         }}   }
   int currentTime = 0;
   for (int i = 0; i < n; i++) {
      if (p[i].AT > currentTime) {
         currentTime = p[i].AT;
      }
      p[i].CT = currentTime + p[i].BT;
      currentTime = p[i].CT;

      p[i].TAT = p[i].CT - p[i].AT;
      p[i].WT = p[i].TAT - p[i].BT;

      totalWT += p[i].WT;
      totalTAT += p[i].TAT;
   }

   // Display and calculate averages
   printf("\nPID\tAT\tBT\tCT\tTAT\tWT\n");
   for (int i = 0; i < n; i++) {
      printf("P%d\t%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].AT, p[i].BT, p[i].CT, p[i].TAT, p[i].WT);
   }

   printf("\nAvg WT = %.2f", (float)totalWT / n);
   printf("\nAvg TAT = %.2f\n", (float)totalTAT / n);

   return 0;}
```

non preemptive sjf;

```
#include <stdio.h>

struct Process {
   int pid; AT,B,  CT, TAT, WT;          // Waiting Time
};

void calculateTimes(struct Process p[], int n) {
   int totalWT = 0, totalTAT = 0;
   int currentTime = 0;
   for (int i = 0; i < n; i++) {
      if (p[i].AT > currentTime) {
         currentTime = p[i].AT;  // Wait for the process to arrive
      }
      p[i].CT = currentTime + p[i].BT;
      currentTime = p[i].CT;
      p[i].TAT = p[i].CT - p[i].AT;
      p[i].WT = p[i].TAT - p[i].BT;
      totalWT += p[i].WT;
      totalTAT += p[i].TAT;
   }
   printf("\nPID\tAT\tBT\tCT\tTAT\tWT\n");
   for (int i = 0; i < n; i++) {
      printf("P%d\t%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].AT, p[i].BT, p[i].CT, p[i].TAT, p[i].WT);
   }
  printf("\nAvg WT = %.2f", (float)totalWT / n);
   printf("\nAvg TAT = %.2f\n", (float)totalTAT / n);
}
void sortProcesses(struct Process p[], int n) {
   // Simple Bubble Sort based on Arrival Time and Burst Time
   for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
         if (p[i].AT > p[j].AT || (p[i].AT == p[j].AT && p[i].BT > p[j].BT)) {
            struct Process temp = p[i];
            p[i] = p[j];
            p[j] = temp;}}}}
int main() {
   int n;
   printf("Enter number of processes: ");
   scanf("%d", &n);
   struct Process p[n];
   // Input Arrival Time and Burst Time for each process
   for (int i = 0; i < n; i++) {
      p[i].pid = i + 1;
      printf("Enter Arrival Time and Burst Time for process %d: ", p[i].pid);
      scanf("%d %d", &p[i].AT, &p[i].BT);
   }
   // Sort processes by Arrival Time
   sortProcesses(p, n);
  // Calculate completion, turnaround, and waiting times
   calculateTimes(p, n);
   return 0;
}
```

Preemptive sjf:

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

struct Process {
 pid,AT BT,    int remainingBT, CT, TAT, WT;             // Waiting Time
};

void calculatePreemptiveSJF(struct Process p[], int n) {
   int totalWT = 0, totalTAT = 0;
   int currentTime = 0, completed = 0;
   bool isCompleted[n];

   for (int i = 0; i < n; i++) {
      p[i].remainingBT = p[i].BT;
      isCompleted[i] = false;
   }
  while (completed < n) {
      int idx = -1;
      int minBT = 9999; // A large number
      // Find the process with the shortest remaining time
      for (int i = 0; i < n; i++) {
         if (p[i].AT <= currentTime && !isCompleted[i] && p[i].remainingBT < minBT) {
            minBT = p[i].remainingBT;
            idx = i;}}
      // If no process is found, increment current time
      if (idx == -1) {
         currentTime++;
         continue;}
      // Execute the selected process
      p[idx].remainingBT--;
      // If process is completed
      if (p[idx].remainingBT == 0) {
         isCompleted[idx] = true;
         completed++;
         p[idx].CT = currentTime + 1; // Set completion time
         p[idx].TAT = p[idx].CT - p[idx].AT; // Calculate turnaround time
         p[idx].WT = p[idx].TAT - p[idx].BT; // Calculate waiting time

         totalWT += p[idx].WT;
         totalTAT += p[idx].TAT;
      }
      currentTime++;
   // Print process details
   printf("\nPID\tAT\tBT\tCT\tTAT\tWT\n");
   for (int i = 0; i < n; i++) {
      printf("P%d\t%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].AT, p[i].BT, p[i].CT, p[i].TAT, p[i].WT);
   }
   printf("\nAvg WT = %.2f", (float)totalWT / n);
   printf("\nAvg TAT = %.2f\n", (float)totalTAT / n);
}
void sortProcesses(struct Process p[], int n) {
   // Simple Bubble Sort based on Arrival Time
```

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

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

    struct Process p[n];

    // Input Arrival Time and Burst Time for each process
    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("Enter Arrival Time and Burst Time for process %d: ", p[i].pid);
        scanf("%d %d", &p[i].AT, &p[i].BT);
    }

    // Sort processes by Arrival Time
    sortProcesses(p, n);

    // Calculate completion, turnaround, and waiting times
    calculatePreemptiveSJF(p, n);

    return 0;
}
```

Round robin:

```
#include <stdio.h>

struct Process {
    int pid;      // Process ID
    int AT;       // Arrival Time
    int BT;       // Burst Time
    int WT;       // Waiting Time
    int TAT;      // Turnaround Time
    int remainingBT; // Remaining Burst Time
};
void calculateRR(struct Process p[], int n, int quantum) {
    int totalWT = 0, totalTAT = 0;
    int currentTime = 0;
    int completed = 0;

    // Initialize remaining burst time
```

```
    for (int i = 0; i < n; i++) {
        p[i].remainingBT = p[i].BT;
        p[i].WT = 0;
        p[i].TAT = 0;
    }
    while (completed < n) {
        int found = 0;
        for (int i = 0; i < n; i++) {
            // Check if the process has arrived and has remaining time
            if (p[i].remainingBT > 0 && p[i].AT <= currentTime) {
                found = 1;
                if (p[i].remainingBT > quantum) {
                    // Execute the process for a time quantum
                    currentTime += quantum;
                    p[i].remainingBT -= quantum;
                } else {
                    // Process finishes
                    currentTime += p[i].remainingBT;
                    p[i].WT = currentTime - p[i].AT - p[i].BT; // Calculate waiting time
                    p[i].TAT = currentTime - p[i].AT; // Calculate turnaround time
                    totalWT += p[i].WT;
                    totalTAT += p[i].TAT;
                    p[i].remainingBT = 0; // Process is complete
                    completed++;}}  }
        // If no process was found, increment time
        if (!found) {
            currentTime++;
        }  }
    // Print process details
    printf("\nPID\tAT\tBT\tTAT\tWT\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].AT, p[i].BT, p[i].TAT, p[i].WT);
    }
    printf("\nAvg WT = %.2f", (float)totalWT / n);
    printf("\nAvg TAT = %.2f\n", (float)totalTAT / n);
}
void sortProcesses(struct Process p[], int n) {
    // Simple Bubble Sort based on Arrival Time
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (p[i].AT > p[j].AT) {
                struct Process temp = p[i];
                p[i] = p[j];
                p[j] = temp;}}}}
int main() {
    int n, quantum;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    struct Process p[n];
```

```
    // Input Arrival Time and Burst Time for each process
    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("Enter Arrival Time and Burst Time for process %d: ", p[i].pid);
        scanf("%d %d", &p[i].AT, &p[i].BT);
    }
    // Sort processes by Arrival Time
    sortProcesses(p, n);
    printf("Enter Time Quantum: ");
    scanf("%d", &quantum);
    // Calculate turnaround and waiting times
    calculateRR(p, n, quantum);
    return 0;
}
```

Fifo place replacement alg:

```
#include <stdio.h>

#define MAX_PAGES 100

void fifoPageReplacement(int pages[], int numPages, int numFrames) {
    int frame[numFrames];
    int pageFaults = 0;
    int k = 0; // For FIFO replacement

    // Initialize frames
    for (int i = 0; i < numFrames; i++) {
        frame[i] = -1; // -1 indicates an empty frame
    }

    printf("\n%-15s %-10s %-15s\n", "Page Reference", "Frame State", "Page Fault");
    printf("%-15s %-10s %-15s\n", "---------------", "-----------", "----------");

    // Process each page reference
    for (int i = 0; i < numPages; i++) {
        int pageFound = 0;

        // Check if the page is already in one of the frames
        for (int j = 0; j < numFrames; j++) {
            if (frame[j] == pages[i]) {
                pageFound = 1; // Page is already in memory
                break;}       }
  // If page is not found in any frame, we have a page fault
        if (!pageFound) {
            frame[k] = pages[i]; // Replace the oldest page
```

```
            k = (k + 1) % numFrames; // Move to the next frame index
            pageFaults++;
        }

        // Display the page reference and current frame state
        printf("%-15d %-10s ", pages[i], "[");
        for (int j = 0; j < numFrames; j++) {
            if (frame[j] != -1) {
                printf("%d", frame[j]);
            } else {
                printf(" ");
            }
            if (j < numFrames - 1) {
                printf(", ");
            }
        }
        printf("] ");

        // Indicate if there was a page fault or a hit
        if (!pageFound) {
            printf("Page Fault\n");
        } else {
            printf("Page Hit\n");
        }
    }

    printf("\nTotal Page Faults: %d\n", pageFaults);
}

int main() {
    int pages[MAX_PAGES], numPages, numFrames;

    // Input number of pages
    printf("Enter number of pages: ");
    scanf("%d", &numPages);

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

    // Input number of frames
    printf("Enter number of frames: ");
    scanf("%d", &numFrames);
    // Calculate FIFO page replacement
    fifoPageReplacement(pages, numPages, numFrames);

    return 0;}
```

LRU page replacement alg:

```
#include <stdio.h>

#define MAX_PAGES 100

void lruPageReplacement(int pages[], int numPages, int numFrames) {
   int frame[numFrames];
   int pageFaults = 0;
   // Initialize frames
   for (int i = 0; i < numFrames; i++) {
      frame[i] = -1; // -1 indicates an empty frame
   }
   printf("\n%-15s %-10s %-15s\n", "Page Reference", "Frame State", "Page Fault");
   printf("%-15s %-10s %-15s\n", "---------------", "-----------", "----------");
   for (int i = 0; i < numPages; i++) {
      int pageFound = 0;
      int lruIndex = -1;
      int minTime = -1;
      // Check if the page is already in one of the frames
      for (int j = 0; j < numFrames; j++) {
         if (frame[j] == pages[i]) {
            pageFound = 1; // Page is already in memory
            break;
         }}
      // If page is not found in any frame, we have a page fault
      if (!pageFound) {
         pageFaults++;
         // Find the least recently used page
         for (int j = 0; j < numFrames; j++) {
            if (frame[j] == -1) {
               lruIndex = j; // Empty frame found
               break;
            } else if (minTime == -1) {
               lruIndex = j; // Initialize first full frame as LRU
            }}
         // Replace the least recently used page
         if (lruIndex != -1) {
            frame[lruIndex] = pages[i];
         } else {
            // If no empty frame, replace the least recently used page
            frame[lruIndex] = pages[i];
         }
      }

      // Display the page reference and current frame state
      printf("%-15d %-10s ", pages[i], "[");
      for (int j = 0; j < numFrames; j++) {
         if (frame[j] != -1) {
```

```
            printf("%d", frame[j]);
        } else {
            printf(" ");}
        if (j < numFrames - 1) {
            printf(", ");
        } }
    printf("] ");

    // Indicate if there was a page fault or a hit
    if (!pageFound) {
        printf("Page Fault\n");
    } else {
        printf("Page Hit\n");
    }
  }

  printf("\nTotal Page Faults: %d\n", pageFaults);
}

int main() {
  int pages[MAX_PAGES], numPages, numFrames;

  // Input number of pages
  printf("Enter number of pages: ");
  scanf("%d", &numPages);

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

  // Input number of frames
  printf("Enter number of frames: ");
  scanf("%d", &numFrames);

  // Calculate LRU page replacement
  lruPageReplacement(pages, numPages, numFrames);

  return 0;
}
```