Lab 5: Implementation of Basic CPU Scheduling

CS363 • Operating System (Lab)

Date: 08 - 09 - 2025

CPU Scheduling in Operating System

CPU scheduling is a process that allows one process to use the CPU while the execution of another process is on hold (in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU. The aim of CPU scheduling is to make the system efficient, fast, and fair.

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects from among the processes in memory that are ready to execute and allocates the CPU to one of them.

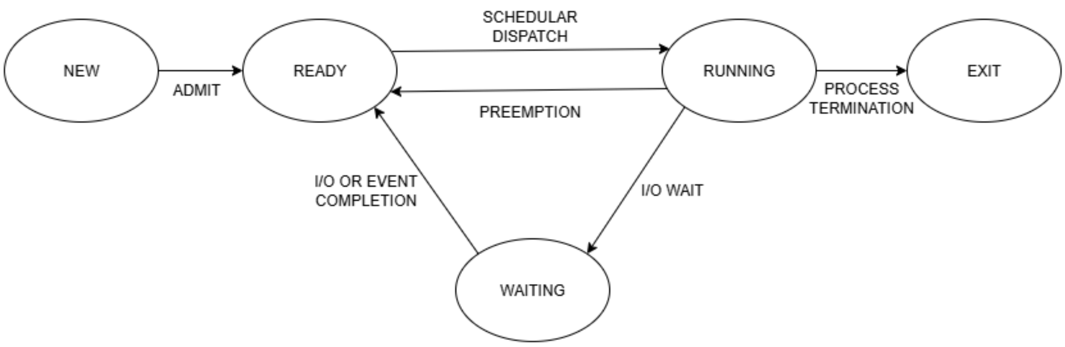

Figure 1: Process state transitions and where scheduling decisions occur.

Types of CPU Scheduling

CPU scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for I/O request or invocation of wait for the termination of one of the child processes).
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs).
3. When a process switches from the waiting state to the ready state (for example, completion of I/O).
4. When a process terminates.

In circumstances 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, in circumstances 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say the scheduling scheme is non-preemptive; otherwise, the scheduling scheme is preemptive.

Non-Preemptive Scheduling

Under non-preemptive scheduling, once the CPU is allocated to a process, that process keeps the CPU until it **voluntarily releases** it by either terminating or blocking for an event/I/O (switching to the waiting state). Only then does the scheduler select the next process from the ready queue.

In non-preemptive schemes, the OS does *not* forcibly interrupt a running process just because another process becomes ready or an I/O completion occurs. Device interrupts are still handled, but after the brief interrupt handler finishes, control returns to the *same* running process.

Common non-preemptive algorithms include: **First-Come, First-Served (FCFS)**, **Shortest Job First (SJF, non-preemptive)**, and **Priority (non-preemptive)**.

Preemptive Scheduling

In this type of scheduling, the tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution.

Thus this type of scheduling is used mainly when a process switches either from running state to ready state or from waiting state to ready state. The resources (that is CPU cycles) are mainly allocated to the process for a limited amount of time and then are taken away, and after that, the process is again placed back in the ready queue in the case if that process still has a CPU burst time remaining. That process stays in the ready queue until it gets the next chance to execute.

Some algorithms that are based on **preemptive** scheduling are **Round Robin (RR)**, **Shortest Remaining Time First (SRTF)**, and **Priority (preemptive version)** scheduling, etc.

CPU Scheduling: Scheduling Criteria

There are several standard criteria used to judge a "good" scheduling algorithm:

1. **CPU Utilization**
   Keep the CPU as *busy as possible*; aim for high utilization.
2. **Throughput**
   Number of processes that *complete* per unit time.
3. **Turnaround Time**
   Total time to execute a process from arrival to completion (wall-clock): $\text{TAT} = CT - AT$.
4. **Waiting Time**
   Total time a process spends in the *ready queue* (excludes CPU and I/O time): $\text{WT} = \text{TAT} - BT \quad \text{(no-I/O model)}$.
5. **Response Time**
   Time from request/arrival until the *first response is produced* (i.e., until the process first gets the CPU; not final output). $\text{RT} = ST - AT$. *Important in time-sharing systems.*

In general, we **maximize** CPU utilization and throughput, and **minimize** turnaround, waiting, and response times (trade-offs may apply).

Notation: AT = arrival time, ST = first start time, CT = completion time, BT = total CPU burst time.

Scheduling Algorithms

To decide which process to execute first and which process to execute last to achieve maximum CPU utilization, computer scientists have defined some algorithms, they are:

- First Come First Serve (FCFS) Scheduling
- Shortest-Job-First (SJF) Scheduling
- Priority Scheduling
- Round Robin (RR) Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling
- Shortest Remaining Time First (SRTF)
- Longest Remaining Time First (LRTF)

First-Come, First-Served (FCFS) Scheduling

The **First-Come, First-Served (FCFS)** scheduling algorithm is the simplest type of CPU scheduling. It is **non-preemptive**, meaning that once the CPU has been allocated to a process, the process keeps it until it either terminates or voluntarily requests I/O. Processes are scheduled in the order in which they arrive in the ready queue.

The implementation of the FCFS policy is easily managed with a **FIFO (First-In, First-Out) queue**. When a process enters the ready queue, its Process Control Block (PCB) is linked to the tail of the queue. When the CPU becomes free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.

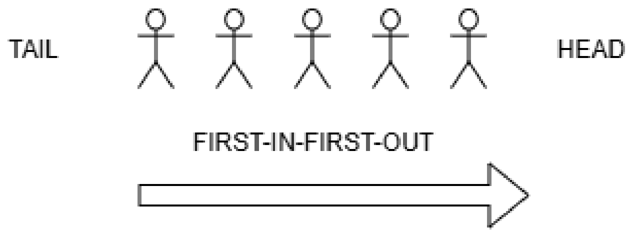

Figure 2: FIFO Queue Representation of FCFS Scheduling

While FCFS is simple and ensures fairness based on arrival order, it suffers from several drawbacks. One major problem is the **convoy effect**, where smaller processes are forced to wait if a long CPU-bound process arrives first. This results in poor utilization and high waiting times for shorter jobs. Moreover, the **average waiting time under FCFS is usually quite high**, especially when there is a large variation in process burst times, making it less suitable for interactive and time-sharing systems.

FCFS Scheduling Program

Listing 1: C Program for FCFS Scheduling

```c

#include <stdio.h>
#include <string.h>

#define MAXN 200
#define MAXSEGS (3*MAXN)
#define MAXLINE 200

typedef struct {
    int id;
    int at;
    int bt;
    int ct;
    int tat;
    int wt;
    int input_idx;
} Proc;

typedef struct {
    char label[16];
    int start;
    int end;
} Segment;

static void stable_sort_by_arrival(Proc p[], int n) {
    int i, j;
    for (i = 1; i < n; ++i) {
        Proc key = p[i];
        j = i - 1;
        while (j >= 0 && (p[j].at > key.at ||
               (p[j].at == key.at && p[j].input_idx > key.input_idx))) {
            p[j + 1] = p[j];
            --j;
        }
        p[j + 1] = key;
    }
}

static int write_repeat(char *buf, int pos, int max, char ch, int count) {
    int k;
    for (k = 0; k < count && pos < max; ++k) buf[pos++] = ch;
    return pos;
}

static int write_centered(char *buf, int pos, int max, const char *text, int w) {
    int len = (int)strlen(text);
    int i;
    if (w < 1) return pos;
    if (len > w - 2) {
        for (i = 0; i < w && pos < max; ++i) buf[pos++] = ' ';
        return pos;
    }
    int left = (w - len) / 2;
    for (i = 0; i < left && pos < max; ++i) buf[pos++] = ' ';
```

```c
    for (i = 0; i < len && pos < max; ++i) buf[pos++] = text[i];
    while ((left + len) < w && pos < max) { buf[pos++] = ' '; ++left; }
    return pos;
}

static void advance_spaces(int *cursor, int n) {
    int s;
    for (s = 0; s < n; ++s) putchar(' ');
    *cursor += n;
}

static void render_gantt(Segment segs[], int segc, int first_time, int last_time) {
    if (segc <= 0 || last_time <= first_time) {
        printf("(no timeline)\n");
        return;
    }

    int total_time = last_time - first_time;
    int scale = 1;
    if (total_time <= 60) scale = 2;
    if (total_time <= 30) scale = 3;
    if (total_time <= 20) scale = 4;
    if (scale > 8) scale = 8;

    char line1[4*MAXLINE];
    int p1 = 0;
    memset(line1, 0, sizeof(line1));

    printf("\n==================== GANTT CHART ====================\n");

    {
        int i;
        for (i = 0; i < segc; ++i) {
            int dur = segs[i].end - segs[i].start;
            int width = dur * scale;
            if (width < 3) width = 3;

            p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '|', 1);

            if (strcmp(segs[i].label, "cs") == 0) {
                p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '-', width);
            } else if (strcmp(segs[i].label, "idle") == 0) {
                p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '.', width);
            } else {
                if (width >= 5) {
                    p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '=', 1);
                    p1 = write_centered(line1, p1, (int)sizeof(line1)-1, segs[i].label, width - 2);
                    p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '=', 1);
                } else {
                    p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '=', width);
                }
```

```c
            }
        }
        p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '|', 1);
        line1[p1] = '\0';
        printf("%s\n", line1);
    }

    {
        int cursor = 0;
        int accum_cols = 0;
        int i;

        for (i = 0; i < segc; ++i) {
            int dur = segs[i].end - segs[i].start;
            int width = dur * scale; if (width < 3) width = 3;

            int boundary_col = accum_cols + i;
            if (boundary_col > cursor) advance_spaces(&cursor, boundary_col - cursor);

            {
                char buf[32]; int len;
                len = snprintf(buf, sizeof(buf), "%d", segs[i].start);
                fputs(buf, stdout);
                cursor += len;
            }
            accum_cols += width;
        }
        {
            int final_boundary_col = accum_cols + segc;
            if (final_boundary_col > cursor) advance_spaces(&cursor, final_boundary_col - cursor);
            {
                char buf[32]; int len;
                len = snprintf(buf, sizeof(buf), "%d", last_time);
                fputs(buf, stdout);
                cursor += len;
            }
        }
        putchar('\n');
    }

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

int main(void) {
    int n, overhead;
    Proc procs[MAXN];
    Segment segs[MAXSEGS];
    int segc = 0;

    printf("Enter number of processes: ");
    if (scanf("%d", &n) != 1 || n <= 0 || n > MAXN) {
        printf("Invalid n.\n");
        return 0;
    }

    printf("Enter context-switch overhead (integer, e.g., 1): ");
```

```c
    if (scanf("%d", &overhead) != 1 || overhead < 0) {
        printf("Invalid overhead.\n");
        return 0;
    }

    {
        int i;
        for (i = 0; i < n; ++i) {
            procs[i].id = i + 1;
            procs[i].input_idx = i;
            printf("Enter Arrival Time and Burst Time for P%d (AT BT): ", i + 1);
            if (scanf("%d %d", &procs[i].at, &procs[i].bt) != 2) {
                printf("Bad input.\n");
                return 0;
            }
            if (procs[i].bt < 0) {
                printf("Burst time must be non-negative.\n");
                return 0;
            }
        }
    }

    stable_sort_by_arrival(procs, n);

    {
        int i;
        int time = 0;
        long long sumBT = 0, total_overhead = 0;

        int minAT = procs[0].at;
        if (time < minAT) {
            strcpy(segs[segc].label, "idle");
            segs[segc].start = time;
            segs[segc].end   = minAT;
            ++segc;
            time = minAT;
        }

        for (i = 0; i < n; ++i) {
            if (time < procs[i].at) {
                strcpy(segs[segc].label, "idle");
                segs[segc].start = time;
                segs[segc].end   = procs[i].at;
                ++segc;
                time = procs[i].at;
            }

            Segment s;
            sprintf(s.label, "P%d", procs[i].id);
            s.start = time;
            s.end   = time + procs[i].bt;
            sumBT   += procs[i].bt;
            procs[i].ct = s.end;
            segs[segc++] = s;
            time = s.end;

            if (i < n - 1) {
```

```c
                if (time >= procs[i + 1].at && overhead > 0) {
                    Segment cso;
                    strcpy(cso.label, "cs");
                    cso.start = time;
                    cso.end   = time + overhead;
                    segs[segc++] = cso;
                    time += overhead;
                    total_overhead += overhead;
                }
            }
        }

        {
            int j;
            long long sumWT = 0, sumTAT = 0;
            for (j = 0; j < n; ++j) {
                procs[j].tat = procs[j].ct - procs[j].at;
                procs[j].wt  = procs[j].tat - procs[j].bt;
                sumWT  += procs[j].wt;
                sumTAT += procs[j].tat;
            }

            {
                int first_arrival = procs[0].at;
                for (j = 1; j < n; ++j)
                    if (procs[j].at < first_arrival) first_arrival = procs[
                        j].at;
                render_gantt(segs, segc, first_arrival, time);
            }

            printf("\n%-11s %-4s %-4s %-4s %-4s %-11s %-4s %-11s\n",
                   "Process ID", "AT", "BT", "CT", "TAT", "(CT-AT)", "WT",
                      "(TAT-BT)");
            for (j = 0; j < n; ++j) {
                char pid[16], tat_expr[32], wt_expr[32];
                snprintf(pid, sizeof(pid), "P%d", procs[j].id);
                snprintf(tat_expr, sizeof(tat_expr), "%d-%d=%d",
                         procs[j].ct, procs[j].at, procs[j].tat);
                snprintf(wt_expr, sizeof(wt_expr), "%d-%d=%d",
                         procs[j].tat, procs[j].bt, procs[j].wt);

                printf("%-11s %-4d %-4d %-4d %-4d %-11s %-4d %-11s\n",
                       pid,
                       procs[j].at, procs[j].bt, procs[j].ct,
                       procs[j].tat, tat_expr,
                       procs[j].wt,  wt_expr);
            }

            printf("\nAverage TAT = %.2f\n", (double)sumTAT / n);
            printf("Average WT  = %.2f\n", (double)sumWT / n);

            {
                int k;
                long long idle_time = 0;
                for (k = 0; k < segc; ++k) {
                    if (strcmp(segs[k].label, "idle") == 0) {
                        int s = segs[k].start < procs[0].at ? procs[0].at :
                            segs[k].start;
```

```c
                        int e = segs[k].end;
                        if (e > s) idle_time += (e - s);
                    }
                }
                {
                    double total_elapsed = (double)(time - procs[0].at);
                    double efficiency = total_elapsed > 0 ? (double)sumBT / total_elapsed * 100.0 : 100.0;

                    printf("\nUseful CPU time (sum BT) = %lld\n", sumBT);
                    printf("Total overhead time      = %lld\n", total_overhead);
                    printf("Total idle time (>=ATmin)= %lld\n", idle_time);
                    printf("Total elapsed (from ATmin= %d to end= %d) = %.0f\n",
                           procs[0].at, time, total_elapsed);
                    printf("Efficiency (Utilization) = %.2f%%\n", efficiency);
                }
            }
        }
    }

    return 0;
}
```

Expected Output:

```
[202463010@paramshavak ~]$ nano fcfs.c
[202463010@paramshavak ~]$ gcc -std=c99 -Wall -Wextra -O2 fcfs.c -o fcfs
[202463010@paramshavak ~]$ ./fcfs
Enter number of processes: 5
Enter context-switch overhead (integer, e.g., 1): 1
Enter Arrival Time and Burst Time for P1 (AT BT): 4 5
Enter Arrival Time and Burst Time for P2 (AT BT): 6 4
Enter Arrival Time and Burst Time for P3 (AT BT): 0 3
Enter Arrival Time and Burst Time for P4 (AT BT): 6 2
Enter Arrival Time and Burst Time for P5 (AT BT): 5 4

==================== GANTT CHART ====================
|=  P3   =|...|=     P1      =|---|=    P5    =|---|=    P2    =|---|= P4 =|
0         3   4               9   10          14  15          19  20     22
=====================================================

Process ID  AT   BT   CT   TAT  (CT-AT)    WT   (TAT-BT)
P3          0    3    3    3    3-0=3      0    3-3=0
P1          4    5    9    5    9-4=5      0    5-5=0
P5          5    4    14   9    14-5=9     5    9-4=5
P2          6    4    19   13   19-6=13    9    13-4=9
P4          6    2    22   16   22-6=16    14   16-2=14

Average TAT = 9.20
Average WT  = 5.60
```

```
Useful CPU time (sum BT) = 18
Total overhead time      = 3
Total idle time (>=ATmin)= 1
Total elapsed (from ATmin= 0 to end= 22) = 22
Efficiency (Utilization) = 81.82%
```

Shortest-Job-First (SJF) Scheduling

The **Shortest-Job-First (SJF)** scheduling algorithm selects the process with the shortest next CPU burst time for execution. It is considered an optimal scheduling algorithm because it minimizes the average waiting time for a given set of processes. However, the main difficulty is that the length of the next CPU burst is not known in advance, and in practice, it is usually predicted based on the recent history of a process's CPU bursts.

Non-Preemptive SJF

In **Non-Preemptive SJF**, once the CPU has been allocated to a process, it cannot be taken away until the process either terminates or moves to the waiting state. The scheduler simply picks the process with the smallest burst time from the ready queue at the time of allocation. This ensures fairness among short jobs, but longer jobs may suffer from starvation if short jobs keep arriving continuously.

Preemptive SJF (Shortest Remaining Time First, SRTF)

In **Preemptive SJF**, also called **Shortest Remaining Time First (SRTF)**, the CPU can be preempted if a new process arrives with a CPU burst smaller than the remaining time of the currently running process. This leads to better response times in interactive systems, as shorter jobs are quickly executed. However, the overhead of frequent preemptions and the possibility of starvation for longer processes remain key drawbacks.

SJF Scheduling Program

Listing 2: C Program for SJF Scheduling (Preemptive & Non-Preemptive)

```c

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

#define MAXN 200
#define MAXSEGS (4*MAXN)
#define MAXLINE 256

typedef struct {
    int id;
    int at, bt;
    int ct, tat, wt;
    int rem;
    int done;
} Proc;

typedef struct {
    char label[16];
    int  start, end;
```

```
} Segment;

static int write_repeat(char *buf, int pos, int max, char ch, int count){
    for (int k = 0; k < count && pos < max; ++k) buf[pos++] = ch;
    return pos;
}

static int write_centered(char *buf, int pos, int max, const char *txt, int
     w){
    int len = (int)strlen(txt);
    if (w < 1) return pos;
    if (len >= w) {
        for (int i=0; i<w && pos<max; ++i) buf[pos++]=txt[i];
        return pos;
    }
    int left = (w - len)/2;
    for (int i=0; i<left && pos<max; ++i) buf[pos++]=' ';
    for (int i=0; i<len && pos<max; ++i) buf[pos++]=txt[i];
    while ((left+len) < w && pos<max){ buf[pos++]=' '; ++left; }
    return pos;
}

static void advance_spaces(int *cursor, int n){
    for (int i=0;i<n;++i) putchar(' ');
    *cursor += n;
}

static void render_gantt(Segment segs[], int segc, int first_time, int
    last_time){
    if (segc <= 0 || last_time <= first_time){
        printf("(no timeline)\n");
        return;
    }

    int total = last_time - first_time;
    int scale = 1;
    if (total <= 60) scale = 2;
    if (total <= 30) scale = 3;
    if (total <= 20) scale = 4;
    if (scale > 8) scale = 8;

    char line1[4*MAXLINE]; int p1 = 0;
    memset(line1, 0, sizeof(line1));

    printf("\n==================== GANTT CHART ====================\n");
    for (int i=0;i<segc;++i){
        int dur = segs[i].end - segs[i].start;
        int w = dur*scale; if (w < 3) w = 3;
        p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '|', 1);

        if (strcmp(segs[i].label,"idle")==0){
            p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '.', w);
        }else{
            if (w >= 5){
                p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '=', 1);
                p1 = write_centered(line1, p1, (int)sizeof(line1)-1, segs[i
                    ].label, w-2);
                p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '=', 1);
```

```
            }else{
                p1 = write_centered(line1, p1, (int)sizeof(line1)-1, segs[i].label, w);
            }
        }
    }
    p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '|', 1);
    line1[p1] = '\0';
    printf("%s\n", line1);

    int cursor = 0, accum = 0;
    for (int i=0;i<segc;++i){
        int dur = segs[i].end - segs[i].start;
        int w = dur*scale; if (w < 3) w = 3;
        int boundary_col = accum + i;
        if (boundary_col > cursor) advance_spaces(&cursor, boundary_col - cursor);
        char buf[32]; int len = snprintf(buf, sizeof(buf), "%d", segs[i].start);
        fputs(buf, stdout); cursor += len;
        accum += w;
    }
    int final_col = accum + segc;
    if (final_col > cursor) advance_spaces(&cursor, final_col - cursor);
    { char buf[32]; int len = snprintf(buf, sizeof(buf), "%d", last_time);
      fputs(buf, stdout); cursor += len; }
    putchar('\n');
    printf("=====================================================\n");
}

static void push_segment(Segment segs[], int *segc, const char *label, int s, int e){
    if (s >= e) return;
    if (*segc > 0){
        Segment *prev = &segs[*segc - 1];
        if (strcmp(prev->label, label)==0 && prev->end == s){
            prev->end = e; return;
        }
    }
    strncpy(segs[*segc].label, label, sizeof(segs[*segc].label)-1);
    segs[*segc].label[sizeof(segs[*segc].label)-1] = '\0';
    segs[*segc].start = s; segs[*segc].end = e;
    (*segc)++;
}

static void print_table(Proc p[], int n){
    double avgTAT=0, avgWT=0;
    printf("\n%-10s %-4s %-4s %-4s %-4s %-10s %-4s %-10s\n",
           "ProcessID","AT","BT","CT","TAT","(CT-AT)","WT","(TAT-BT)");
    for (int i=0;i<n;++i){
        char pid[16], tatExpr[32], wtExpr[32];
        snprintf(pid,sizeof(pid),"P%d",p[i].id);
        snprintf(tatExpr,sizeof(tatExpr),"%d-%d=%d",p[i].ct,p[i].at,p[i].tat);
        snprintf(wtExpr,sizeof(wtExpr),"%d-%d=%d",p[i].tat,p[i].bt,p[i].wt);
        avgTAT += p[i].tat; avgWT += p[i].wt;
        printf("%-10s %-4d %-4d %-4d %-4d %-10s %-4d %-10s\n",
```

```
                pid, p[i].at, p[i].bt, p[i].ct, p[i].tat,
                tatExpr, p[i].wt, wtExpr);
    }
    printf("\nAverage TAT = %.2f\n", avgTAT/n);
    printf("Average WT  = %.2f\n", avgWT/n);
}

static void sjf_nonpreemptive(Proc p[], int n){
    int done=0, t=0, first_arr=INT_MAX, last_ct=0;
    long long sumBT=0;
    Segment segs[MAXSEGS]; int segc=0;
    for (int i=0;i<n;++i){ if(p[i].at<first_arr)first_arr=p[i].at; sumBT+=p
        [i].bt; p[i].done=0; }
    if (t<first_arr) t=first_arr;

    while(done<n){
        int idx=-1,minBT=INT_MAX;
        for(int i=0;i<n;++i){
            if(!p[i].done && p[i].at<=t){
                if(p[i].bt<minBT || (p[i].bt==minBT && p[i].at<p[idx].at)){
                    minBT=p[i].bt; idx=i;
                }
            }
        }
        if(idx==-1){
            int nextAT=INT_MAX;
            for(int i=0;i<n;++i) if(!p[i].done && p[i].at>t && p[i].at<
                nextAT) nextAT=p[i].at;
            push_segment(segs,&segc,"idle",t,nextAT);
            t=nextAT; continue;
        }
        char label[16]; snprintf(label,sizeof(label),"P%d",p[idx].id);
        push_segment(segs,&segc,label,t,t+p[idx].bt);
        t+=p[idx].bt; p[idx].ct=t; p[idx].done=1; last_ct=t; done++;
    }

    for(int i=0;i<n;++i){ p[i].tat=p[i].ct-p[i].at; p[i].wt=p[i].tat-p[i].
        bt; }
    render_gantt(segs,segc,first_arr,last_ct);
    print_table(p,n);
    double total_elapsed=(double)(last_ct-first_arr);
    double efficiency=total_elapsed>0?(double)sumBT/total_elapsed
        *100.0:100.0;
    printf("\nUseful CPU time (sum BT) = %lld\n",sumBT);
    printf("Total elapsed (from ATmin= %d to end= %d) = %.0f\n",first_arr,
        last_ct,total_elapsed);
    printf("Efficiency (Utilization) = %.2f%%\n",efficiency);
}

static void sjf_preemptive(Proc p[], int n){
    int done=0,t=0,first_arr=INT_MAX,last_ct=0;
    long long sumBT=0;
    Segment segs[MAXSEGS]; int segc=0;
    for(int i=0;i<n;++i){p[i].rem=p[i].bt;p[i].done=0;if(p[i].at<first_arr)
        first_arr=p[i].at;sumBT+=p[i].bt;}
    if(t<first_arr) t=first_arr;

    int current=-1,seg_start=t;
```

```
    while(done<n){
        int idx=-1,minR=INT_MAX;
        for(int i=0;i<n;++i){
            if(p[i].at<=t && p[i].rem>0){
                if(p[i].rem<minR || (p[i].rem==minR && p[i].at<p[idx].at)){
                    minR=p[i].rem; idx=i;
                }
            }
        }
        if(idx==-1){
            if(current!=-1){
                char label[16];snprintf(label,sizeof(label),"P%d",p[current].id);
                push_segment(segs,&segc,label,seg_start,t); current=-1;
            }
            int nextAT=INT_MAX;
            for(int i=0;i<n;++i) if(p[i].rem>0&&p[i].at>t&&p[i].at<nextAT) nextAT=p[i].at;
            push_segment(segs,&segc,"idle",t,nextAT); t=nextAT; seg_start=t; continue;
        }
        if(current!=idx){
            if(current!=-1){
                char label[16];snprintf(label,sizeof(label),"P%d",p[current].id);
                push_segment(segs,&segc,label,seg_start,t);
            }
            current=idx; seg_start=t;
        }
        p[current].rem--; t++;
        if(p[current].rem==0){
            p[current].ct=t; p[current].done=1; done++;
            char label[16];snprintf(label,sizeof(label),"P%d",p[current].id);
            push_segment(segs,&segc,label,seg_start,t);
            current=-1; seg_start=t; if(t>last_ct)last_ct=t;
        }
    }

    for(int i=0;i<n;++i){p[i].tat=p[i].ct-p[i].at;p[i].wt=p[i].tat-p[i].bt; if(p[i].ct>last_ct)last_ct=p[i].ct;}
    render_gantt(segs,segc,first_arr,last_ct);
    print_table(p,n);
    double total_elapsed=(double)(last_ct-first_arr);
    double efficiency=total_elapsed>0?(double)sumBT/total_elapsed*100.0:100.0;
    printf("\nUseful CPU time (sum BT) = %lld\n",sumBT);
    printf("Total elapsed (from ATmin= %d to end= %d) = %.0f\n",first_arr,last_ct,total_elapsed);
    printf("Efficiency (Utilization) = %.2f%%\n",efficiency);
}

int main(void){
    int n,choice; Proc p[MAXN];
    printf("Select Scheduling Type:\n1. Preemptive SJF (SRTF)\n2. Non-Preemptive SJF\nChoice: ");
    if(scanf("%d",&choice)!=1||(choice!=1&&choice!=2)){printf("Invalid choice.\n");return 0;}
```

```c
    printf("Enter number of processes: "); if(scanf("%d",&n)!=1||n<=0||n>MAXN){printf("Invalid n.\n");return 0;}
    for(int i=0;i<n;++i){p[i].id=i+1;printf("Enter Arrival Time and Burst Time for P%d (AT BT): ",i+1);
        if(scanf("%d %d",&p[i].at,&p[i].bt)!=2||p[i].bt<0){printf("Bad input.\n");return 0;}
        p[i].ct=p[i].tat=p[i].wt=0;p[i].rem=p[i].bt;p[i].done=0;}
    if(choice==2) sjf_nonpreemptive(p,n); else sjf_preemptive(p,n);
    return 0;
}
```

Expected Output (Non-Preemptive):

```
[202463010@paramshavak ~]$ nano sjf4.c
[202463010@paramshavak ~]$ gcc -std=c99 -Wall -Wextra -O2 sjf4.c -o sjf4
[202463010@paramshavak ~]$ ./sjf4
Select Scheduling Type:
1. Preemptive SJF (SRTF)
2. Non-Preemptive SJF
Choice: 2
Enter number of processes: 4
Enter Arrival Time and Burst Time for P1 (AT BT): 0 6
Enter Arrival Time and Burst Time for P2 (AT BT): 0 8
Enter Arrival Time and Burst Time for P3 (AT BT): 0 7
Enter Arrival Time and Burst Time for P4 (AT BT): 0 3

==================== GANTT CHART ====================
|=  P4   =|=       P1       =|=        P3          =|=          P2           =|
0         3                  9                     16                        24
=====================================================

ProcessID  AT   BT   CT   TAT  (CT-AT)    WT   (TAT-BT)
P1         0    6    9    9    9-0=9      3    9-6=3
P2         0    8    24   24   24-0=24    16   24-8=16
P3         0    7    16   16   16-0=16    9    16-7=9
P4         0    3    3    3    3-0=3      0    3-3=0

Average TAT = 13.00
Average WT  = 7.00

Useful CPU time (sum BT) = 24
Total elapsed (from ATmin= 0 to end= 24) = 24
Efficiency (Utilization) = 100.00%
```

Expected Output (Preemptive):

```
[202463010@paramshavak ~]$ nano sjf4.c
[202463010@paramshavak ~]$ gcc -std=c99 -Wall -Wextra -O2 sjf4.c -o sjf4
[202463010@paramshavak ~]$ ./sjf4
Select Scheduling Type:
1. Preemptive SJF (SRTF)
2. Non-Preemptive SJF
```

```
Choice: 1
Enter number of processes: 4
Enter Arrival Time and Burst Time for P1 (AT BT): 0 8
Enter Arrival Time and Burst Time for P2 (AT BT): 1 4
Enter Arrival Time and Burst Time for P3 (AT BT): 2 9
Enter Arrival Time and Burst Time for P4 (AT BT): 3 5

==================== GANTT CHART ====================
|P1 |=    P2    =|=     P4      =|=        P1         =|=            P3              =|
0   1           5              10                     17                            26
=====================================================

ProcessID  AT   BT   CT   TAT  (CT-AT)    WT   (TAT-BT)
P1         0    8    17   17   17-0=17    9    17-8=9
P2         1    4    5    4    5-1=4      0    4-4=0
P3         2    9    26   24   26-2=24    15   24-9=15
P4         3    5    10   7    10-3=7     2    7-5=2

Average TAT = 13.00
Average WT  = 6.50

Useful CPU time (sum BT) = 26
Total elapsed (from ATmin= 0 to end= 26) = 26
Efficiency (Utilization) = 100.00%
```

NOTE: The remaining CPU scheduling algorithms such as Priority Scheduling, Round Robin (RR) and ..will be covered in the next lab session.