

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.

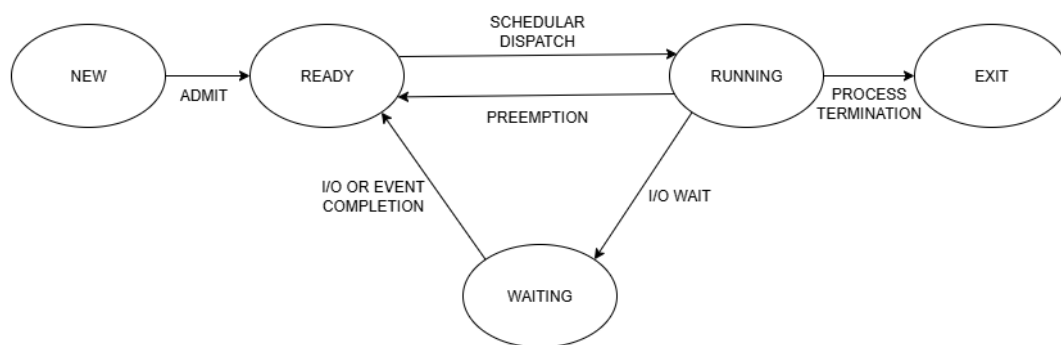


Figure 1: Process state transitions and where scheduling decisions occur.

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): $TAT = CT - AT$.
4. **Waiting Time**
Total time a process spends in the *ready queue* (excludes CPU and I/O time): $WT = TAT - BT$ (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). $RT = ST - AT$. *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.

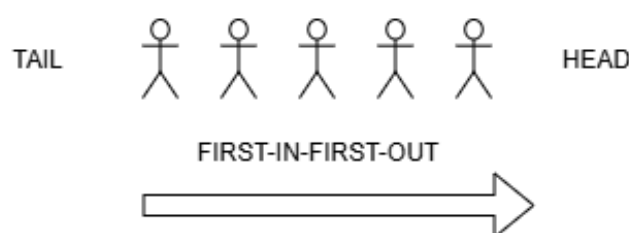


Figure 2: FIFO Queue Representation of FCFS Scheduling

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

```
1
2 #include <stdio.h>
3 #include <string.h>
4
5 #define MAXN 200
6 #define MAXSEGS (3*MAXN)
7 #define MAXLINE 200
8
9 typedef struct {
10     int id;
11     int at;
12     int bt;
13     int ct;
14     int tat;
15     int wt;
16     int input_idx;
17 } Proc;
18
19 typedef struct {
20     char label[16];
21     int start;
22     int end;
23 } Segment;
24
25 static void stable_sort_by_arrival(Proc p[], int n) {
26     int i, j;
27     for (i = 1; i < n; ++i) {
28         Proc key = p[i];
29         j = i - 1;
30         while (j >= 0 && (p[j].at > key.at ||
31             (p[j].at == key.at && p[j].input_idx > key.input_idx))) {
32             p[j + 1] = p[j];
33             --j;
34         }
35         p[j + 1] = key;
36     }
37 }
38
39 static int write_repeat(char *buf, int pos, int max, char ch, int count) {
40     int k;
41     for (k = 0; k < count && pos < max; ++k) buf[pos++] = ch;
42     return pos;
43 }
44
45 static int write_centered(char *buf, int pos, int max, const char *text,
46     int w) {
47     int len = (int)strlen(text);
48     int i;
49     if (w < 1) return pos;
50     if (len > w - 2) {
51         for (i = 0; i < w && pos < max; ++i) buf[pos++] = ' ';
52         return pos;
53     }
54     int left = (w - len) / 2;
55     for (i = 0; i < left && pos < max; ++i) buf[pos++] = ' ';
```

```

55     for (i = 0; i < len && pos < max; ++i) buf[pos++] = text[i];
56     while ((left + len) < w && pos < max) { buf[pos++] = ' '; ++left; }
57     return pos;
58 }
59
60 static void advance_spaces(int *cursor, int n) {
61     int s;
62     for (s = 0; s < n; ++s) putchar(' ');
63     *cursor += n;
64 }
65
66 static void render_gantt(Segment segs[], int segc, int first_time, int
67     last_time) {
68     if (segc <= 0 || last_time <= first_time) {
69         printf("(no timeline)\n");
70         return;
71     }
72     int total_time = last_time - first_time;
73     int scale = 1;
74     if (total_time <= 60) scale = 2;
75     if (total_time <= 30) scale = 3;
76     if (total_time <= 20) scale = 4;
77     if (scale > 8) scale = 8;
78
79     char line1[4*MAXLINE];
80     int p1 = 0;
81     memset(line1, 0, sizeof(line1));
82
83     printf("\n===== GANTT CHART =====\n");
84
85     {
86         int i;
87         for (i = 0; i < segc; ++i) {
88             int dur = segs[i].end - segs[i].start;
89             int width = dur * scale;
90             if (width < 3) width = 3;
91
92             p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '|', 1);
93
94             if (strcmp(segs[i].label, "cs") == 0) {
95                 p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '-',
96                     width);
97             } else if (strcmp(segs[i].label, "idle") == 0) {
98                 p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '.',
99                     width);
100             } else {
101                 if (width >= 5) {
102                     p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '=',
103                         1);
104                     p1 = write_centered(line1, p1, (int)sizeof(line1)-1,
105                         segs[i].label, width - 2);
106                     p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '=',
107                         1);
108                 } else {
109                     p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '=',
110                         width);
111                 }
112             }
113         }
114     }

```

```

106     }
107 }
108 p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '|', 1);
109 line1[p1] = '\0';
110 printf("%s\n", line1);
111 }
112
113 {
114     int cursor = 0;
115     int accum_cols = 0;
116     int i;
117
118     for (i = 0; i < segc; ++i) {
119         int dur = segs[i].end - segs[i].start;
120         int width = dur * scale; if (width < 3) width = 3;
121
122         int boundary_col = accum_cols + i;
123         if (boundary_col > cursor) advance_spaces(&cursor, boundary_col
            - cursor);
124
125         {
126             char buf[32]; int len;
127             len = snprintf(buf, sizeof(buf), "%d", segs[i].start);
128             fputs(buf, stdout);
129             cursor += len;
130         }
131         accum_cols += width;
132     }
133     {
134         int final_boundary_col = accum_cols + segc;
135         if (final_boundary_col > cursor) advance_spaces(&cursor,
            final_boundary_col - cursor);
136
137         {
138             char buf[32]; int len;
139             len = snprintf(buf, sizeof(buf), "%d", last_time);
140             fputs(buf, stdout);
141             cursor += len;
142         }
143         putchar('\n');
144     }
145
146     printf("=====\n");
147 }
148
149 int main(void) {
150     int n, overhead;
151     Proc procs[MAXN];
152     Segment segs[MAXSEGS];
153     int segc = 0;
154
155     printf("Enter number of processes: ");
156     if (scanf("%d", &n) != 1 || n <= 0 || n > MAXN) {
157         printf("Invalid n.\n");
158         return 0;
159     }
160
161     printf("Enter context-switch overhead (integer, e.g., 1): ");

```

```
162     if (scanf("%d", &overhead) != 1 || overhead < 0) {
163         printf("Invalid overhead.\n");
164         return 0;
165     }
166
167     {
168         int i;
169         for (i = 0; i < n; ++i) {
170             procs[i].id = i + 1;
171             procs[i].input_idx = i;
172             printf("Enter Arrival Time and Burst Time for P%d (AT BT): ", i
173                 + 1);
174             if (scanf("%d %d", &procs[i].at, &procs[i].bt) != 2) {
175                 printf("Bad input.\n");
176                 return 0;
177             }
178             if (procs[i].bt < 0) {
179                 printf("Burst time must be non-negative.\n");
180                 return 0;
181             }
182         }
183
184         stable_sort_by_arrival(procs, n);
185
186         {
187             int i;
188             int time = 0;
189             long long sumBT = 0, total_overhead = 0;
190
191             int minAT = procs[0].at;
192             if (time < minAT) {
193                 strcpy(segs[segc].label, "idle");
194                 segs[segc].start = time;
195                 segs[segc].end = minAT;
196                 ++segc;
197                 time = minAT;
198             }
199
200             for (i = 0; i < n; ++i) {
201                 if (time < procs[i].at) {
202                     strcpy(segs[segc].label, "idle");
203                     segs[segc].start = time;
204                     segs[segc].end = procs[i].at;
205                     ++segc;
206                     time = procs[i].at;
207                 }
208
209                 Segment s;
210                 sprintf(s.label, "P%d", procs[i].id);
211                 s.start = time;
212                 s.end = time + procs[i].bt;
213                 sumBT += procs[i].bt;
214                 procs[i].ct = s.end;
215                 segs[segc++] = s;
216                 time = s.end;
217
218                 if (i < n - 1) {
```

```

219         if (time >= procs[i + 1].at && overhead > 0) {
220             Segment cso;
221             strcpy(cso.label, "cs");
222             cso.start = time;
223             cso.end   = time + overhead;
224             segs[segc++] = cso;
225             time += overhead;
226             total_overhead += overhead;
227         }
228     }
229 }
230
231 {
232     int j;
233     long long sumWT = 0, sumTAT = 0;
234     for (j = 0; j < n; ++j) {
235         procs[j].tat = procs[j].ct - procs[j].at;
236         procs[j].wt  = procs[j].tat - procs[j].bt;
237         sumWT  += procs[j].wt;
238         sumTAT += procs[j].tat;
239     }
240
241     {
242         int first_arrival = procs[0].at;
243         for (j = 1; j < n; ++j)
244             if (procs[j].at < first_arrival) first_arrival = procs[
                j].at;
245         render_gantt(segs, segc, first_arrival, time);
246     }
247
248     printf("\n%-11s %-4s %-4s %-4s %-4s %-11s %-4s %-11s\n",
249           "Process ID", "AT", "BT", "CT", "TAT", "(CT-AT)", "WT",
250           "(TAT-BT)");
251     for (j = 0; j < n; ++j) {
252         char pid[16], tat_expr[32], wt_expr[32];
253         snprintf(pid, sizeof(pid), "P%d", procs[j].id);
254         snprintf(tat_expr, sizeof(tat_expr), "%d-%d=%d",
255                 procs[j].ct, procs[j].at, procs[j].tat);
256         snprintf(wt_expr, sizeof(wt_expr), "%d-%d=%d",
257                 procs[j].tat, procs[j].bt, procs[j].wt);
258
259         printf("%-11s %-4d %-4d %-4d %-4d %-11s %-4d %-11s\n",
260               pid,
261               procs[j].at, procs[j].bt, procs[j].ct,
262               procs[j].tat, tat_expr,
263               procs[j].wt, wt_expr);
264     }
265
266     printf("\nAverage TAT = %.2f\n", (double)sumTAT / n);
267     printf("Average WT  = %.2f\n", (double)sumWT / n);
268
269     {
270         int k;
271         long long idle_time = 0;
272         for (k = 0; k < segc; ++k) {
273             if (strcmp(segs[k].label, "idle") == 0) {
                int s = segs[k].start < procs[0].at ? procs[0].at :
                    segs[k].start;

```



```

274         int e = segs[k].end;
275         if (e > s) idle_time += (e - s);
276     }
277 }
278 {
279     double total_elapsed = (double)(time - procs[0].at);
280     double efficiency = total_elapsed > 0 ? (double)sumBT /
        total_elapsed * 100.0 : 100.0;
281
282     printf("\nUseful CPU time (sum BT) = %lld\n", sumBT);
283     printf("Total overhead time      = %lld\n",
        total_overhead);
284     printf("Total idle time (>=ATmin)= %lld\n", idle_time);
285     printf("Total elapsed (from ATmin= %d to end= %d) = %.0
        f\n",
286         procs[0].at, time, total_elapsed);
287     printf("Efficiency (Utilization) = %.2f%%\n",
        efficiency);
288 }
289 }
290 }
291 }
292
293     return 0;
294 }

```

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)

```

1  #include <stdio.h>
2  #include <string.h>
3  #include <limits.h>
4
5
6  #define MAXN 200
7  #define MAXSEGS (4*MAXN)
8  #define MAXLINE 256
9
10 typedef struct {
11     int id;
12     int at, bt;
13     int ct, tat, wt;
14     int rem;
15     int done;
16 } Proc;
17
18 typedef struct {
19     char label[16];
20     int start, end;

```

```

21 } Segment;
22
23 static int write_repeat(char *buf, int pos, int max, char ch, int count){
24     for (int k = 0; k < count && pos < max; ++k) buf[pos++] = ch;
25     return pos;
26 }
27
28 static int write_centered(char *buf, int pos, int max, const char *txt, int
29     w){
30     int len = (int)strlen(txt);
31     if (w < 1) return pos;
32     if (len >= w) {
33         for (int i=0; i<w && pos<max; ++i) buf[pos++] = txt[i];
34         return pos;
35     }
36     int left = (w - len)/2;
37     for (int i=0; i<left && pos<max; ++i) buf[pos++] = ' ';
38     for (int i=0; i<len && pos<max; ++i) buf[pos++] = txt[i];
39     while ((left+len) < w && pos<max){ buf[pos++] = ' '; ++left; }
40     return pos;
41 }
42
43 static void advance_spaces(int *cursor, int n){
44     for (int i=0; i<n; ++i) putchar(' ');
45     *cursor += n;
46 }
47
48 static void render_gantt(Segment segs[], int segc, int first_time, int
49     last_time){
50     if (segc <= 0 || last_time <= first_time){
51         printf("(no timeline)\n");
52         return;
53     }
54
55     int total = last_time - first_time;
56     int scale = 1;
57     if (total <= 60) scale = 2;
58     if (total <= 30) scale = 3;
59     if (total <= 20) scale = 4;
60     if (scale > 8) scale = 8;
61
62     char line1[4*MAXLINE]; int p1 = 0;
63     memset(line1, 0, sizeof(line1));
64
65     printf("\n===== GANTT CHART =====\n");
66     for (int i=0; i<segc; ++i){
67         int dur = segs[i].end - segs[i].start;
68         int w = dur*scale; if (w < 3) w = 3;
69         p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '|', 1);
70
71         if (strcmp(segs[i].label, "idle")==0){
72             p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '.', w);
73         }else{
74             if (w >= 5){
75                 p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '=', 1);
76                 p1 = write_centered(line1, p1, (int)sizeof(line1)-1, segs[i]
77                     .label, w-2);
78                 p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '=', 1);

```

```

76         }else{
77             p1 = write_centered(line1, p1, (int)sizeof(line1)-1, segs[i]
78                 .label, w);
79         }
80     }
81     p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '|', 1);
82     line1[p1] = '\0';
83     printf("%s\n", line1);
84
85     int cursor = 0, accum = 0;
86     for (int i=0;i<segc;++i){
87         int dur = segs[i].end - segs[i].start;
88         int w = dur*scale; if (w < 3) w = 3;
89         int boundary_col = accum + i;
90         if (boundary_col > cursor) advance_spaces(&cursor, boundary_col -
91             cursor);
92         char buf[32]; int len = snprintf(buf, sizeof(buf), "%d", segs[i].
93             start);
94         fputs(buf, stdout); cursor += len;
95         accum += w;
96     }
97     int final_col = accum + segc;
98     if (final_col > cursor) advance_spaces(&cursor, final_col - cursor);
99     { char buf[32]; int len = snprintf(buf, sizeof(buf), "%d", last_time);
100         fputs(buf, stdout); cursor += len; }
101     putchar('\n');
102     printf("=====\n");
103 }
104
105 static void push_segment(Segment segs[], int *segc, const char *label, int
106     s, int e){
107     if (s >= e) return;
108     if (*segc > 0){
109         Segment *prev = &segs[*segc - 1];
110         if (strcmp(prev->label, label)==0 && prev->end == s){
111             prev->end = e; return;
112         }
113     }
114     strncpy(segs[*segc].label, label, sizeof(segs[*segc].label)-1);
115     segs[*segc].label[sizeof(segs[*segc].label)-1] = '\0';
116     segs[*segc].start = s; segs[*segc].end = e;
117     (*segc)++;
118 }
119
120 static void print_table(Proc p[], int n){
121     double avgTAT=0, avgWT=0;
122     printf("\n%-10s %-4s %-4s %-4s %-4s %-10s %-4s %-10s\n",
123         "ProcessID", "AT", "BT", "CT", "TAT", "(CT-AT)", "WT", "(TAT-BT)");
124     for (int i=0;i<n;++i){
125         char pid[16], tatExpr[32], wtExpr[32];
126         snprintf(pid, sizeof(pid), "%d", p[i].id);
127         snprintf(tatExpr, sizeof(tatExpr), "%d-%d=%d", p[i].ct, p[i].at, p[i].
128             tat);
129         snprintf(wtExpr, sizeof(wtExpr), "%d-%d=%d", p[i].tat, p[i].bt, p[i].wt)
130             ;
131         avgTAT += p[i].tat; avgWT += p[i].wt;
132         printf("%-10s %-4d %-4d %-4d %-4d %-10s %-4d %-10s\n",

```

```

128         pid, p[i].at, p[i].bt, p[i].ct, p[i].tat,
129         tatExpr, p[i].wt, wtExpr);
130     }
131     printf("\nAverage TAT = %.2f\n", avgTAT/n);
132     printf("Average WT = %.2f\n", avgWT/n);
133 }
134
135 static void sjf_nonpreemptive(Proc p[], int n){
136     int done=0, t=0, first_arr=INT_MAX, last_ct=0;
137     long long sumBT=0;
138     Segment segs[MAXSEGS]; int segc=0;
139     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; }
140     if (t<first_arr) t=first_arr;
141
142     while(done<n){
143         int idx=-1,minBT=INT_MAX;
144         for(int i=0;i<n;++i){
145             if(!p[i].done && p[i].at<=t){
146                 if(p[i].bt<minBT || (p[i].bt==minBT && p[i].at<p[idx].at)){
147                     minBT=p[i].bt; idx=i;
148                 }
149             }
150         }
151         if(idx!=-1){
152             int nextAT=INT_MAX;
153             for(int i=0;i<n;++i) if(!p[i].done && p[i].at>t && p[i].at<
                nextAT) nextAT=p[i].at;
154             push_segment(segs,&segc,"idle",t,nextAT);
155             t=nextAT; continue;
156         }
157         char label[16]; sprintf(label,sizeof(label),"P%d",p[idx].id);
158         push_segment(segs,&segc,label,t,t+p[idx].bt);
159         t+=p[idx].bt; p[idx].ct=t; p[idx].done=1; last_ct=t; done++;
160     }
161
162     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; }
163     render_gantt(segs,segc,first_arr,last_ct);
164     print_table(p,n);
165     double total_elapsed=(double)(last_ct-first_arr);
166     double efficiency=total_elapsed>0?(double)sumBT/total_elapsed
        *100.0:100.0;
167     printf("\nUseful CPU time (sum BT) = %lld\n",sumBT);
168     printf("Total elapsed (from ATmin= %d to end= %d) = %.0f\n",first_arr,
        last_ct,total_elapsed);
169     printf("Efficiency (Utilization) = %.2f%%\n",efficiency);
170 }
171
172 static void sjf_preemptive(Proc p[], int n){
173     int done=0,t=0,first_arr=INT_MAX,last_ct=0;
174     long long sumBT=0;
175     Segment segs[MAXSEGS]; int segc=0;
176     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;}
177     if(t<first_arr) t=first_arr;
178
179     int current=-1,seg_start=t;

```

```

180 while(done<n){
181     int idx=-1,minR=INT_MAX;
182     for(int i=0;i<n;++i){
183         if(p[i].at<=t && p[i].rem>0){
184             if(p[i].rem<minR || (p[i].rem==minR && p[i].at<p[idx].at)){
185                 minR=p[i].rem; idx=i;
186             }
187         }
188     }
189     if(idx==-1){
190         if(current!=-1){
191             char label[16];snprintf(label,sizeof(label),"P%d",p[current
192                 ].id);
193             push_segment(segs,&segc,label,seg_start,t); current=-1;
194         }
195         int nextAT=INT_MAX;
196         for(int i=0;i<n;++i) if(p[i].rem>0&&p[i].at>t&&p[i].at<nextAT)
197             nextAT=p[i].at;
198         push_segment(segs,&segc,"idle",t,nextAT); t=nextAT; seg_start=t
199         ; continue;
200     }
201     if(current!=idx){
202         if(current!=-1){
203             char label[16];snprintf(label,sizeof(label),"P%d",p[current
204                 ].id);
205             push_segment(segs,&segc,label,seg_start,t);
206         }
207         current=idx; seg_start=t;
208     }
209     p[current].rem--; t++;
210     if(p[current].rem==0){
211         p[current].ct=t; p[current].done=1; done++;
212         char label[16];snprintf(label,sizeof(label),"P%d",p[current].id
213             );
214         push_segment(segs,&segc,label,seg_start,t);
215         current=-1; seg_start=t; if(t>last_ct)last_ct=t;
216     }
217 }
218
219 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;
220     if(p[i].ct>last_ct)last_ct=p[i].ct;}
221 render_gantt(segs,segc,first_arr,last_ct);
222 print_table(p,n);
223 double total_elapsed=(double)(last_ct-first_arr);
224 double efficiency=total_elapsed>0?(double)sumBT/total_elapsed
225     *100.0:100.0;
226 printf("\nUseful CPU time (sum BT) = %lld\n",sumBT);
227 printf("Total elapsed (from ATmin= %d to end= %d) = %.0f\n",first_arr,
228     last_ct,total_elapsed);
229 printf("Efficiency (Utilization) = %.2f%%\n",efficiency);
230 }
231
232 int main(void){
233     int n,choice; Proc p[MAXN];
234     printf("Select Scheduling Type:\n1. Preemptive SJF (SRTF)\n2. Non-
235         Preemptive SJF\nChoice: ");
236     if(scanf("%d",&choice)!=1|| (choice!=1&&choice!=2)){printf("Invalid
237         choice.\n");return 0;}

```

```

228     printf("Enter number of processes: "); if(scanf("%d",&n)!=1||n<=0||n>
        MAXN){printf("Invalid n.\n");return 0;}
229     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);
230         if(scanf("%d %d",&p[i].at,&p[i].bt)!=2||p[i].bt<0){printf("Bad
            input.\n");return 0;}
231         p[i].ct=p[i].tat=p[i].wt=0;p[i].rem=p[i].bt;p[i].done=0;}
232     if(choice==2) sjf_nonpreemptive(p,n); else sjf_preemptive(p,n);
233     return 0;
234 }

```

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%
```


Priority Scheduling

Priority Scheduling is a CPU scheduling algorithm in which each process is assigned a *priority value*. The CPU is always allocated to the process with the *highest priority*. In case two processes have the same priority, they are scheduled according to the **First-Come, First-Served (FCFS)** policy.

Interestingly, the **Shortest Job First (SJF)** scheduling algorithm can be considered as a special case of Priority Scheduling, where the priority is inversely proportional to the predicted CPU burst time: shorter jobs get higher priority.

Non-Preemptive Priority Scheduling

In the non-preemptive version, once the CPU has been allocated to a process, it cannot be taken away until the process either terminates or voluntarily enters the waiting state (e.g., for I/O). If a new process with a higher priority arrives, it must wait in the ready queue until the current process finishes. This approach is simple to implement but may result in delayed execution of urgent tasks.

Preemptive Priority Scheduling

In the preemptive version, the CPU may be preempted if a new process with a higher priority arrives while another process is executing. This ensures responsiveness for critical tasks and is often used in real-time systems. However, it increases context switching overhead.

Problem: Starvation

A significant drawback of priority scheduling is the problem of **starvation**, also known as **indefinite blocking**. A low-priority process may remain waiting indefinitely if higher-priority processes continue to arrive, thereby monopolizing the CPU.

Solution: Aging

The solution to starvation is a technique called **aging**. Aging gradually increases the priority of processes that have been waiting in the ready queue for a long time. For example, if priorities range from 127 (lowest) to 0 (highest), the priority of a waiting process could be improved by 1 every 15 minutes. Eventually, even processes with initially low priority will gain enough priority to execute, ensuring fairness in the system.

Priority Scheduling Program

Listing 3: C Program for Priority Scheduling (Preemptive & Non-Preemptive)

```

1  #include <stdio.h>
2  #include <string.h>
3  #include <limits.h>
4
5  #define MAXN 200
6  #define MAXSEGS (6*MAXN)
7  #define MAXLINE 256
8
9  typedef struct {
10     int id;
11     int at, bt, pr;
12     int ct, tat, wt;
13     int rem;
14     int done;
15 } Proc;
16
17 typedef struct {
18     char label[16];
19     int start, end;
20 } Segment;
21
22 static int write_repeat(char *buf, int pos, int max, char ch, int count){
23     for (int k = 0; k < count && pos < max; ++k) buf[pos++] = ch;
24     return pos;
25 }
26
27 static int write_centered(char *buf, int pos, int max, const char *txt, int
    w){
28     int len = (int)strlen(txt);
29     if (w < 1) return pos;
30     if (len >= w) { for (int i=0; i<w && pos<max; ++i) buf[pos++]=txt[i];
        return pos; }
31     int left = (w - len)/2;
32     for (int i=0; i<left && pos<max; ++i) buf[pos++]=' ';
33     for (int i=0; i<len && pos<max; ++i) buf[pos++]=txt[i];
34     while ((left+len) < w && pos<max){ buf[pos++]=' '; ++left; }
35     return pos;
36 }
37
38 static void advance_spaces(int *cursor, int n){
39     for (int i=0; i<n; ++i) putchar(' ');
40     *cursor += n;
41 }
42
43 static void render_gantt(Segment segs[], int segc, int first_time, int
    last_time){
44     if (segc <= 0 || last_time <= first_time){
45         printf("(no timeline)\n");
46         return;
47     }
48
49     int total = last_time - first_time;
50     int scale = 1;
51     if (total <= 60) scale = 2;
52     if (total <= 30) scale = 3;

```

```

53     if (total <= 20) scale = 4;
54     if (scale > 8) scale = 8;
55
56     char line1[4*MAXLINE]; int p1 = 0;
57     memset(line1, 0, sizeof(line1));
58
59     printf("\n===== GANTT CHART =====\n");
60     for (int i=0;i<segc;++i){
61         int dur = segs[i].end - segs[i].start;
62         int w = dur*scale; if (w < 3) w = 3;
63         p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '|', 1);
64
65         if (strcmp(segs[i].label,"idle")==0){
66             p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '.', w);
67         }else{
68             if (w >= 5){
69                 p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '=', 1);
70                 p1 = write_centered(line1, p1, (int)sizeof(line1)-1, segs[i]
71                                     ].label, w-2);
72                 p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '=', 1);
73             }else{
74                 p1 = write_centered(line1, p1, (int)sizeof(line1)-1, segs[i]
75                                     ].label, w);
76             }
77         }
78         p1 = write_repeat(line1, p1, (int)sizeof(line1)-1, '|', 1);
79         line1[p1] = '\0';
80         printf("%s\n", line1);
81
82         int cursor = 0, accum = 0;
83         for (int i=0;i<segc;++i){
84             int dur = segs[i].end - segs[i].start;
85             int w = dur*scale; if (w < 3) w = 3;
86             int boundary_col = accum + i;
87             if (boundary_col > cursor) advance_spaces(&cursor, boundary_col -
88                                                     cursor);
89             char buf[32]; int len = snprintf(buf, sizeof(buf), "%d", segs[i].
90                                             start);
91             fputs(buf, stdout); cursor += len;
92             accum += w;
93         }
94         int final_col = accum + segc;
95         if (final_col > cursor) advance_spaces(&cursor, final_col - cursor);
96         { char buf[32]; int len = snprintf(buf, sizeof(buf), "%d", last_time);
97           fputs(buf, stdout); cursor += len; }
98         putchar('\n');
99         printf("=====\n");
100     }
101
102     static void push_segment(Segment segs[], int *segc, const char *label, int
103                             s, int e){
104         if (s >= e) return;
105         if (*segc > 0){
106             Segment *prev = &segs[*segc - 1];
107             if (strcmp(prev->label, label)==0 && prev->end == s){
108                 prev->end = e; return;
109             }
110         }

```

```

106     }
107     strncpy(segs[*segc].label, label, sizeof(segs[*segc].label)-1);
108     segs[*segc].label[sizeof(segs[*segc].label)-1] = '\0';
109     segs[*segc].start = s; segs[*segc].end = e;
110     (*segc)++;
111 }
112
113 static void print_table(Proc p[], int n){
114     double avgTAT=0, avgWT=0;
115     printf("\n%-10s %-4s %-4s %-4s %-4s %-10s %-4s %-10s\n",
116           "ProcessID", "AT", "BT", "PR", "CT", "TAT", "(CT-AT)", "WT", "(TAT-BT)");
117
118     for (int i=0; i<n; ++i){
119         char pid[16], tatExpr[32], wtExpr[32];
120         snprintf(pid, sizeof(pid), "P%d", p[i].id);
121         snprintf(tatExpr, sizeof(tatExpr), "%d-%d=%d", p[i].ct, p[i].at, p[i].tat);
122         snprintf(wtExpr, sizeof(wtExpr), "%d-%d=%d", p[i].tat, p[i].bt, p[i].wt);
123
124         avgTAT += p[i].tat; avgWT += p[i].wt;
125         printf("%-10s %-4d %-4d %-4d %-4d %-10s %-4d %-10s\n",
126               pid, p[i].at, p[i].bt, p[i].pr, p[i].ct, p[i].tat,
127               tatExpr, p[i].wt, wtExpr);
128     }
129     printf("\nAverage TAT = %.2f\n", avgTAT/n);
130     printf("Average WT = %.2f\n", avgWT/n);
131 }
132
133 static void priority_nonpreemptive(Proc p[], int n){
134     int done=0, t=0, first_arr=INT_MAX, last_ct=0;
135     long long sumBT=0;
136     Segment segs[MAXSEGS]; int segc=0;
137
138     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; }
139     if (t < first_arr) t = first_arr;
140
141     while(done<n){
142         int idx=-1, bestPr=INT_MAX;
143         for(int i=0; i<n; ++i){
144             if(!p[i].done && p[i].at<=t){
145                 if(p[i].pr<bestPr || (p[i].pr==bestPr && p[i].at<p[idx].at))
146                     bestPr=p[i].pr; idx=i;
147             }
148         }
149         if(idx!=-1){
150             int nextAT=INT_MAX;
151             for(int i=0; i<n; ++i) if(!p[i].done && p[i].at>t && p[i].at<nextAT) nextAT=p[i].at;
152             push_segment(segs, &segc, "idle", t, nextAT);
153             t=nextAT; continue;
154         }
155         char label[16]; snprintf(label, sizeof(label), "P%d", p[idx].id);
156         push_segment(segs, &segc, label, t, t+p[idx].bt);
157         t += p[idx].bt;
158         p[idx].ct = t; p[idx].done=1; last_ct=t; done++;
159     }

```

```

158     }
159
160     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; }
161     render_gantt(segs,segc,first_arr,last_ct);
162     print_table(p,n);
163
164     double total_elapsed = (double)(last_ct - first_arr);
165     double efficiency = total_elapsed>0 ? (double)sumBT/total_elapsed*100.0
        : 100.0;
166     printf("\nUseful CPU time (sum BT) = %lld\n", sumBT);
167     printf("Total elapsed (from ATmin= %d to end= %d) = %.0f\n", first_arr,
        last_ct, total_elapsed);
168     printf("Efficiency (Utilization) = %.2f%%\n", efficiency);
169 }
170
171 static void priority_preemptive(Proc p[], int n){
172     int done=0, t=0, first_arr=INT_MAX, last_ct=0;
173     long long sumBT=0;
174     Segment segs[MAXSEGS]; int segc=0;
175
176     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; }
177     if (t < first_arr) t = first_arr;
178
179     int current=-1, seg_start=t;
180
181     while(done<n){
182         int idx=-1, bestPr=INT_MAX;
183         for(int i=0;i<n;++i){
184             if(p[i].at<=t && p[i].rem>0){
185                 if(p[i].pr<bestPr || (p[i].pr==bestPr && p[i].at<p[idx].at)
                    ){
186                     bestPr=p[i].pr; idx=i;
187                 }
188             }
189         }
190
191         if(idx==-1){
192             if(current!=-1){
193                 char lab[16]; snprintf(lab,sizeof(lab),"P%d",p[current].id)
                    ;
194                 push_segment(segs,&segc,lab,seg_start,t);
195                 current=-1;
196             }
197             int nextAT=INT_MAX;
198             for(int i=0;i<n;++i) if(p[i].rem>0 && p[i].at>t && p[i].at<
                nextAT) nextAT=p[i].at;
199             push_segment(segs,&segc,"idle",t,nextAT);
200             t=nextAT; seg_start=t; continue;
201         }
202
203         if(current!=idx){
204             if(current!=-1){
205                 char lab[16]; snprintf(lab,sizeof(lab),"P%d",p[current].id)
                    ;
206                 push_segment(segs,&segc,lab,seg_start,t);
207             }

```

```

208         current=idx; seg_start=t;
209     }
210
211     p[current].rem--; t++;
212     if(p[current].rem==0){
213         p[current].ct=t; p[current].done=1; done++;
214         char lab[16]; snprintf(lab,sizeof(lab),"P%d",p[current].id);
215         push_segment(segs,&segc,lab,seg_start,t);
216         current=-1; seg_start=t; if(t>last_ct) last_ct=t;
217     }
218 }
219
220 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; }
221 render_gantt(segs,segc,first_arr,last_ct);
222 print_table(p,n);
223
224 double total_elapsed = (double)(last_ct - first_arr);
225 double efficiency = total_elapsed>0 ? (double)sumBT/total_elapsed*100.0
    : 100.0;
226 printf("\nUseful CPU time (sum BT) = %lld\n", sumBT);
227 printf("Total elapsed (from ATmin= %d to end= %d) = %.0f\n", first_arr,
    last_ct, total_elapsed);
228 printf("Efficiency (Utilization) = %.2f%%\n", efficiency);
229 }
230
231 int main(void){
232     int n, choice; Proc p[MAXN];
233
234     printf("Select Scheduling Type:\n");
235     printf("1. Preemptive Priority\n");
236     printf("2. Non-Preemptive Priority\n");
237     printf("Choice: ");
238     if (scanf("%d",&choice)!=1 || (choice!=1 && choice!=2)){ printf("
        Invalid choice.\n"); return 0; }
239
240     printf("Enter number of processes: ");
241     if (scanf("%d",&n)!=1 || n<=0 || n>MAXN){ printf("Invalid n.\n");
        return 0; }
242
243     printf("(Note: lower PR value means higher priority; e.g., 0 is highest
        )\n");
244     for(int i=0;i<n;++i){
245         p[i].id=i+1;
246         printf("Enter AT, BT, PR for P%d: ", i+1);
247         if (scanf("%d %d %d",&p[i].at,&p[i].bt,&p[i].pr)!=3 || p[i].bt<0){
248             printf("Bad input.\n"); return 0;
249         }
250         p[i].ct=p[i].tat=p[i].wt=0; p[i].rem=p[i].bt; p[i].done=0;
251     }
252
253     if (choice==2) priority_nonpreemptive(p,n);
254     else          priority_preemptive(p,n);
255
256     return 0;
257 }

```

Expected Output (Non-Preemptive):

```
[202463010@paramshavak ~]$ nano ps.c
[202463010@paramshavak ~]$ gcc -std=c99 -Wall -Wextra -O2 ps.c -o ps
[202463010@paramshavak ~]$ ./ps
Select Scheduling Type:
1. Preemptive Priority
2. Non-Preemptive Priority
Choice: 2
Enter number of processes: 4
(Note: lower PR value means higher priority; e.g., 0 is highest)
Enter AT, BT, PR for P1: 0 6 2
Enter AT, BT, PR for P2: 4 10 1
Enter AT, BT, PR for P3: 4 4 2
Enter AT, BT, PR for P4: 8 3 1

===== GANTT CHART =====
|=      P1      |=      P2      |=  P4  |=  P3  |=
0              6              16      19      23
=====

ProcessID  AT   BT   PR   CT   TAT  (CT-AT)   WT   (TAT-BT)
P1         0    6    2    6    6    6-0=6     0    6-6=0
P2         4   10    1   16   12   16-4=12    2   12-10=2
P3         4    4    2   23   19   23-4=19   15   19-4=15
P4         8    3    1   19   11   19-8=11    8   11-3=8

Average TAT = 12.00
Average WT  = 6.25

Useful CPU time (sum BT) = 23
Total elapsed (from ATmin= 0 to end= 23) = 23
Efficiency (Utilization) = 100.00%
```

Expected Output (Preemptive):

```
[202463010@paramshavak ~]$ nano ps.c
[202463010@paramshavak ~]$ gcc -std=c99 -Wall -Wextra -O2 ps.c -o ps
[202463010@paramshavak ~]$ ./ps
Select Scheduling Type:
1. Preemptive Priority
2. Non-Preemptive Priority
Choice: 1
Enter number of processes: 5
(Note: lower PR value means higher priority; e.g., 0 is highest)
Enter AT, BT, PR for P1: 0 4 4
Enter AT, BT, PR for P2: 1 3 3
Enter AT, BT, PR for P3: 2 1 2
Enter AT, BT, PR for P4: 3 5 1
Enter AT, BT, PR for P5: 4 2 1
```

===== GANTT CHART =====

```
| P1 | P2 | P3 | =          P4          = | P5 = | P2 = | P1 = |
0   1   2   3                   8       10       12       15
```

=====

ProcessID	AT	BT	PR	CT	TAT	(CT-AT)	WT	(TAT-BT)
P1	0	4	4	15	15	15-0=15	11	15-4=11
P2	1	3	3	12	11	12-1=11	8	11-3=8
P3	2	1	2	3	1	3-2=1	0	1-1=0
P4	3	5	1	8	5	8-3=5	0	5-5=0
P5	4	2	1	10	6	10-4=6	4	6-2=4

Average TAT = 7.60

Average WT = 4.60

Useful CPU time (sum BT) = 15

Total elapsed (from ATmin= 0 to end= 15) = 15

Efficiency (Utilization) = 100.00%

Round Robin (RR) Scheduling

Round Robin (RR) is a preemptive CPU scheduling algorithm designed especially for time-sharing systems. It is similar to the First-Come, First-Served (FCFS) scheduling algorithm, but adds *preemption* through the use of a fixed *time quantum* (or *time slice*), generally ranging from 10 to 100 milliseconds.

In RR, the ready queue is managed in the same FIFO manner as shown in Figure 2, but conceptually arranged as a **circular queue**. Each process is allocated CPU time for a maximum of one time quantum. If a process does not finish within this time, it is preempted and placed at the *tail of the ready queue*. The CPU scheduler then allocates the CPU to the next process at the head of the queue.

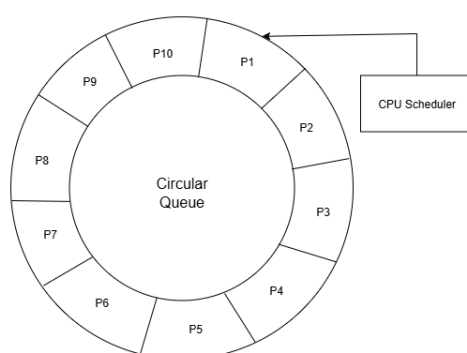


Figure 3: Circular Queue Representation of Round Robin Scheduling

The execution flow of Round Robin scheduling is shown in Figure 4. It illustrates how the scheduler decides whether a process will complete within its time quantum or be preempted and returned to the ready queue.

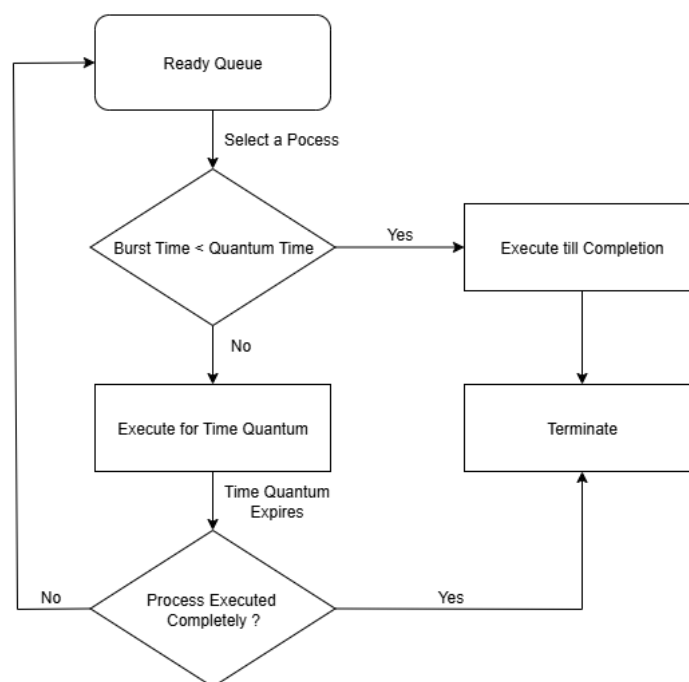


Figure 4: Flowchart of Round Robin Scheduling Execution

Choice of Time Quantum.

- If the time quantum is very large, RR degenerates into FCFS scheduling.
- If the time quantum is very small, frequent context switches lead to high *overhead* and low CPU efficiency.
- An ideal time quantum is large compared to the context switch time but small enough to ensure good responsiveness.

Round Robin improves fairness and responsiveness by ensuring that no process can monopolize the CPU for long. It significantly reduces the *response time* for interactive processes, though the average turnaround and waiting times may vary depending on the time quantum and process mix.

Round Robin Scheduling Program

Listing 4: C Program for Round Robin

```

1  #include <stdio.h>
2  #include <string.h>
3  #include <limits.h>
4
5  #define MAXN      200
6  #define MAXSEGS  (8*MAXN)
7  #define MAXLINE  256
8
9  typedef struct {
10     int id;
11     int at, bt;
12     int ct, tat, wt;
13     int rem;
14 } Proc;
15
16 typedef struct {
17     char label[16];
18     int start, end;
19 } Segment;
20
21 typedef struct {
22     int data[4*MAXN];
23     int head, tail;
24 } Queue;
25
26 static void q_init(Queue *q){ q->head = q->tail = 0; }
27 static int q_empty(Queue *q){ return q->head == q->tail; }
28 static void q_push(Queue *q, int v){ q->data[q->tail++] = v; }
29 static int q_pop (Queue *q){ return q->data[q->head++]; }
30
31 static int write_repeat(char *buf, int pos, int max, char ch, int count){
32     for (int k = 0; k < count && pos < max; ++k) buf[pos++] = ch;
33     return pos;
34 }
35 static int write_centered(char *buf, int pos, int max, const char *txt, int
    w){
36     int len = (int)strlen(txt);
37     if (w < 1) return pos;

```

```

38     if (len >= w){ for (int i=0;i<w && pos<max;++i) buf[pos++] = txt[i];
39         return pos; }
40     int left = (w - len)/2;
41     for (int i=0; i<left && pos<max; ++i) buf[pos++]=' ';
42     for (int i=0; i<len && pos<max; ++i) buf[pos++] = txt[i];
43     while ((left+len) < w && pos<max){ buf[pos++]=' '; ++left; }
44     return pos;
45 }
46 static void advance_spaces(int *cursor, int n){
47     for (int i=0;i<n;++i) putchar(' ');
48     *cursor += n;
49 }
50 static void push_segment(Segment segs[], int *segc, const char *label, int
51 s, int e){
52     if (s >= e) return;
53     if (*segc > 0){
54         Segment *prev = &segs[*segc - 1];
55         if (strcmp(prev->label,label)==0 && prev->end==s){ prev->end = e;
56             return; }
57     }
58     strncpy(segs[*segc].label,label,sizeof(segs[*segc].label)-1);
59     segs[*segc].label[sizeof(segs[*segc].label)-1] = '\0';
60     segs[*segc].start = s; segs[*segc].end = e;
61     (*segc)++;
62 }
63 static void render_gantt(Segment segs[], int segc, int first_time, int
64 last_time){
65     if (segc<=0 || last_time<=first_time){ printf("(no timeline)\n");
66         return; }
67
68     int total = last_time - first_time;
69     int scale = 1;
70     if (total <= 60) scale = 2;
71     if (total <= 30) scale = 3;
72     if (total <= 20) scale = 4;
73     if (scale > 8) scale = 8;
74
75     char line1[4*MAXLINE]; int p1 = 0; memset(line1,0,sizeof(line1));
76
77     printf("\n===== GANTT CHART =====\n");
78     for (int i=0;i<segc;++i){
79         int dur = segs[i].end - segs[i].start;
80         int w = dur*scale; if (w < 3) w = 3;
81         p1 = write_repeat(line1,p1,(int)sizeof(line1)-1,'|',1);
82         if (strcmp(segs[i].label,"idle")==0){
83             p1 = write_repeat(line1,p1,(int)sizeof(line1)-1,'.',w);
84         }else{
85             if (w >= 5){
86                 p1 = write_repeat(line1,p1,(int)sizeof(line1)-1,'=',1);
87                 p1 = write_centered(line1,p1,(int)sizeof(line1)-1,segs[i].
88                     label,w-2);
89                 p1 = write_repeat(line1,p1,(int)sizeof(line1)-1,'=',1);
90             }else{
91                 p1 = write_centered(line1,p1,(int)sizeof(line1)-1,segs[i].
92                     label,w);
93             }
94         }
95     }
96 }

```

```

89  p1 = write_repeat(line1,p1,(int)sizeof(line1)-1,'|',1);
90  line1[p1] = '\0';
91  printf("%s\n", line1);
92
93  int cursor=0, accum=0;
94  for (int i=0;i<segc;++i){
95      int dur = segs[i].end - segs[i].start;
96      int w    = dur*scale; if (w < 3) w = 3;
97      int boundary_col = accum + i;
98      if (boundary_col > cursor) advance_spaces(&cursor, boundary_col -
99          cursor);
100     char buf[32]; int len = snprintf(buf,sizeof(buf),"d",segs[i].start
101         );
102     fputs(buf, stdout); cursor += len;
103     accum += w;
104 }
105 int final_col = accum + segc;
106 if (final_col > cursor) advance_spaces(&cursor, final_col - cursor);
107 { char buf[32]; int len = snprintf(buf,sizeof(buf),"d",last_time);
108     fputs(buf,stdout); cursor+=len; }
109 putchar('\n');
110 printf("=====\n");
111 }
112
113 static void print_table(Proc p[], int n){
114     double avgTAT=0, avgWT=0;
115     printf("\n%-10s %-4s %-4s %-4s %-4s %-10s %-4s %-10s\n",
116         "ProcessID", "AT", "BT", "CT", "TAT", "(CT-AT)", "WT", "(TAT-BT)");
117     for (int i=0;i<n;++i){
118         char pid[16], tatExpr[32], wtExpr[32];
119         snprintf(pid,sizeof(pid),"P%d",p[i].id);
120         snprintf(tatExpr,sizeof(tatExpr),"d-%d=%d",p[i].ct,p[i].at,p[i].
121             tat);
122         snprintf(wtExpr,sizeof(wtExpr),"d-%d=%d",p[i].tat,p[i].bt,p[i].wt)
123             ;
124         avgTAT += p[i].tat; avgWT += p[i].wt;
125         printf("%-10s %-4d %-4d %-4d %-4d %-10s %-4d %-10s\n",
126             pid, p[i].at, p[i].bt, p[i].ct, p[i].tat,
127             tatExpr, p[i].wt, wtExpr);
128     }
129     printf("\nAverage TAT = %.2f\n", avgTAT/n);
130     printf("Average WT  = %.2f\n", avgWT/n);
131 }
132
133 int main(void){
134     int n, q;
135     Proc p[MAXN];
136     Segment segs[MAXSEGS]; int segc=0;
137
138     printf("Enter time quantum q: ");
139     if (scanf("%d",&q)!=1 || q<=0){ printf("Invalid quantum.\n"); return 0;
140         }
141
142     printf("Enter number of processes: ");
143     if (scanf("%d",&n)!=1 || n<=0 || n>MAXN){ printf("Invalid n.\n");
144         return 0; }
145
146     for (int i=0;i<n;++i){

```

```

140     p[i].id = i+1;
141     printf("Enter Arrival Time and Burst Time for P%d (AT BT): ", i+1);
142     if (scanf("%d %d",&p[i].at,&p[i].bt)!=2 || p[i].bt<0){ printf("Bad
        input.\n"); return 0; }
143     p[i].ct = p[i].tat = p[i].wt = 0;
144     p[i].rem = p[i].bt;
145 }
146
147 long long sumBT=0; for(int i=0;i<n;++i) sumBT += p[i].bt;
148 int first_arr=INT_MAX, last_ct=0;
149 for (int i=0;i<n;++i) if (p[i].at < first_arr) first_arr = p[i].at;
150
151 int t = first_arr;
152 int enqed[MAXN]={0};
153 Queue rq; q_init(&rq);
154
155 for (int i=0;i<n;++i) if (!enqed[i] && p[i].at<=t && p[i].rem>0){
        q_push(&rq,i); enqed[i]=1; }
156
157 int done=0;
158 while (done < n){
159     if (q_empty(&rq)){
160         int nextAT=INT_MAX;
161         for (int i=0;i<n;++i)
162             if (p[i].rem>0 && p[i].at>t && p[i].at<nextAT) nextAT=p[i].
                at;
163         push_segment(segs,&segc,"idle",t,nextAT);
164         t = nextAT;
165         for (int i=0;i<n;++i) if (!enqed[i] && p[i].at<=t && p[i].rem
                >0){ q_push(&rq,i); enqed[i]=1; }
166         continue;
167     }
168
169     int idx = q_pop(&rq);
170     int slice = p[idx].rem < q ? p[idx].rem : q;
171     char lab[16]; snprintf(lab,sizeof(lab),"P%d",p[idx].id);
172     push_segment(segs,&segc,lab, t, t+slice);
173     t += slice;
174     p[idx].rem -= slice;
175
176     for (int i=0;i<n;++i)
177         if (!enqed[i] && p[i].at<=t && p[i].rem>0){ q_push(&rq,i);
                enqed[i]=1; }
178
179     if (p[idx].rem > 0){
180         q_push(&rq, idx);
181     }else{
182         p[idx].ct = t;
183         if (t > last_ct) last_ct = t;
184         done++;
185     }
186 }
187
188 for (int i=0;i<n;++i){
189     p[i].tat = p[i].ct - p[i].at;
190     p[i].wt = p[i].tat - p[i].bt;
191     if (p[i].ct > last_ct) last_ct = p[i].ct;
192 }

```

```

193     render_gantt(segs, segc, first_arr, last_ct);
194     print_table(p, n);
195
196     double total_elapsed = (double)(last_ct - first_arr);
197     double efficiency = total_elapsed > 0 ? (double)sumBT / total_elapsed * 100.0
198         : 100.0;
199
200     printf("\nTime quantum (q)           = %d\n", q);
201     printf("Useful CPU time (sum BT) = %lld\n", sumBT);
202     printf("Total elapsed (from ATmin= %d to end= %d) = %.0f\n", first_arr,
203         last_ct, total_elapsed);
204     printf("Efficiency (Utilization) = %.2f%%\n", efficiency);
205     return 0;
206 }

```

Expected Output:

```

[202463010@paramshavak ~]$ nano rr.c
[202463010@paramshavak ~]$ gcc -std=c99 -Wall -Wextra -O2 rr.c -o rr
[202463010@paramshavak ~]$ ./rr

```

Enter time quantum q: 2

Enter number of processes: 3

Enter Arrival Time and Burst Time for P1 (AT BT): 0 5

Enter Arrival Time and Burst Time for P2 (AT BT): 4 2

Enter Arrival Time and Burst Time for P3 (AT BT): 5 4

```

===== GANTT CHART =====
|=      P1      |=|= P2 |=| P1 |=      P3      |=
0              4          6   7              11
=====

```

ProcessID	AT	BT	CT	TAT	(CT-AT)	WT	(TAT-BT)
P1	0	5	7	7	7-0=7	2	7-5=2
P2	4	2	6	2	6-4=2	0	2-2=0
P3	5	4	11	6	11-5=6	2	6-4=2

Average TAT = 5.00

Average WT = 1.33

Time quantum (q) = 2

Useful CPU time (sum BT) = 11

Total elapsed (from ATmin= 0 to end= 11) = 11

Efficiency (Utilization) = 100.00%

ASSIGNMENT NOTICE: The assignment for this lab will be given by the TAs during the lab session.