```c
/*
    COMP3511 Fall 2024
    PA2: Multi-level Feedback Queue

    Your name: LI,Yuntong
    Your ITSC email: ylino@connect.ust.hk

    Declaration:

    I declare that I am not involved in plagiarism
    I understand that both parties (i.e., students providing the codes and student

*/

// Note: Necessary header files are included
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

// Define MAX_NUM_PROCESS
// For simplicity, assume that we have at most 10 processes
#define MAX_NUM_PROCESS 10
#define MAX_PROCESS_NAME 5
#define MAX_GANTT_CHART 300

// N-level Feedback Queue (N=1,2,3,4)
#define MAX_NUM_QUEUE 4

// Keywords (to be used when parsing the input)
#define KEYWORD_QUEUE_NUMBER "queue_num"
#define KEYWORD_TQ "time_quantum"
#define KEYWORD_PROCESS_TABLE_SIZE "process_table_size"
#define KEYWORD_PROCESS_TABLE "process_table"

// Assume that we only need to support 2 types of space characters:
// " " (space), "\t" (tab)
#define SPACE_CHARS " \t"

// Process data structure
// Helper functions:
//  process_init: initialize a process entry
//  process_table_print: Display the process table
struct Process {
    char name[MAX_PROCESS_NAME];
    int arrival_time ;
    int burst_time;
    int remain_time; // remain_time is needed in the intermediate steps of MLFQ
};
void process_init(struct Process* p, char name[MAX_PROCESS_NAME], int arrival_time, int bu
    strcpy(p->name, name);
    p->arrival_time = arrival_time;
    p->burst_time = burst_time;
    p->remain_time = 0;
}
void process_table_print(struct Process* p, int size) {
    int i;
    printf("Process\tArrival\tBurst\n");
    for (i=0; i<size; i++) {
```

```c
        printf("%s\t%d\t%d\n", p[i].name, p[i].arrival_time, p[i].burst_time);
    }
}

//This is the helpful functions created by me
struct Queue {
    int values[MAX_NUM_PROCESS];
    int front, rear, count;
};
void queue_init(struct Queue* q) {
    q->count = 0;
    q->front = 0;
    q->rear = -1;
}
int queue_is_empty(struct Queue* q) {
    return q->count == 0;
}
int queue_is_full(struct Queue* q) {
    return q->count == MAX_NUM_PROCESS;
}

int queue_peek(struct Queue* q) {
    return q->values[q->front];
}
void queue_enqueue(struct Queue* q, int new_value) {
    if (!queue_is_full(q)) {
        if ( q->rear == MAX_NUM_PROCESS -1)
            q->rear = -1;
        q->values[++q->rear] = new_value;
        q->count++;
        //printf("Enqueued %d, front: %d, rear: %d, count: %d\n", new_value, q->front, q->
    }
}
void queue_dequeue(struct Queue* q) {
    //printf("Dequeued %d, front: %d, rear: %d, count: %d\n", q->values[q->front], q->
    q->front++;
    if (q->front == MAX_NUM_PROCESS)
        q->front = 0;
    q->count--;
}
void queue_print(struct Queue* q) {
    int c = q->count;
    printf("size = %d\n", c);
    int cur = q->front;
    printf("values = ");
    while ( c > 0 ) {
        if ( cur == MAX_NUM_PROCESS )
            cur = 0;
        printf("%d ", q->values[cur]);
        cur++;
        c--;
    }
    printf("\n");
}


// A simple GanttChart structure
// Helper functions:
//   gantt_chart_print: display the current chart
struct GanttChartItem {
```

```
121:     char name[MAX_PROCESS_NAME];
122:     int duration;
123: };
124:
125: void gantt_chart_update(struct GanttChartItem chart[MAX_GANTT_CHART], int* n, char name[MA
126:     int i;
127:     i = *n;
128:     // The new item is the same as the last item
129:     if ( i > 0 && strcmp(chart[i-1].name, name) == 0)
130:     {
131:         chart[i-1].duration += duration; // update duration
132:     }
133:     else
134:     {
135:         strcpy(chart[i].name, name);
136:         chart[i].duration = duration;
137:         *n = i+1;
138:     }
139: }
140:
141: void gantt_chart_print(struct GanttChartItem chart[MAX_GANTT_CHART], int n) {
142:     int t = 0;
143:     int i = 0;
144:     printf("Gantt Chart = ");
145:     printf("%d ", t);
146:     for (i=0; i<n; i++) {
147:         t = t + chart[i].duration;
148:         printf("%s %d ", chart[i].name, t);
149:     }
150:     printf("\n");
151: }
152:
153: // Global variables
154: int queue_num = 0;
155: int process_table_size = 0;
156: struct Process process_table[MAX_NUM_PROCESS];
157: int time_quantum[MAX_NUM_QUEUE];
158:
159:
160: // Helper function: Check whether the line is a blank line (for input parsing)
161: int is_blank(char *line) {
162:     char *ch = line;
163:     while ( *ch != '\0' ) {
164:         if ( !isspace(*ch) )
165:             return 0;
166:         ch++;
167:     }
168:     return 1;
169: }
170: // Helper function: Check whether the input line should be skipped
171: int is_skip(char *line) {
172:     if ( is_blank(line) )
173:         return 1;
174:     char *ch = line ;
175:     while ( *ch != '\0' ) {
176:         if ( !isspace(*ch) && *ch == '#')
177:             return 1;
178:         ch++;
179:     }
180:     return 0;
```

```
181: }
182: // Helper: parse_tokens function
183: void parse_tokens(char **argv, char *line, int *numTokens, char *delimiter) {
184:     int argc = 0;
185:     char *token = strtok(line, delimiter);
186:     while (token != NULL)
187:     {
188:         argv[argc++] = token;
189:         token = strtok(NULL, delimiter);
190:     }
191:     *numTokens = argc;
192: }
193:
194: // Helper: parse the input file
195: void parse_input() {
196:     FILE *fp = stdin;
197:     char *line = NULL;
198:     ssize_t nread;
199:     size_t len = 0;
200:
201:     char *two_tokens[2]; // buffer for 2 tokens
202:     char *queue_tokens[MAX_NUM_QUEUE]; // buffer for MAX_NUM_QUEUE tokens
203:     int n;
204:
205:     int numTokens = 0, i=0;
206:     char equal_plus_spaces_delimiters[5] = "";
207:
208:     char process_name[MAX_PROCESS_NAME];
209:     int process_arrival_time = 0;
210:     int process_burst_time = 0;
211:
212:     strcpy(equal_plus_spaces_delimiters, "=");
213:     strcat(equal_plus_spaces_delimiters,SPACE_CHARS);
214:
215:     // Note: MingGW don't have getline, so you are forced to do the coding in Linux/PO
216:     // In other words, you cannot easily coding in Windows environment
217:
218:     while ( (nread = getline(&line, &len, fp)) != -1 ) {
219:         if ( is_skip(line) == 0)  {
220:             line = strtok(line,"\n");
221:
222:             if (strstr(line, KEYWORD_QUEUE_NUMBER)) {
223:                 // parse queue_num
224:                 parse_tokens(two_tokens, line, &numTokens, equal_plus_spaces_delimiters);
225:                 if (numTokens == 2) {
226:                     sscanf(two_tokens[1], "%d", &queue_num);
227:                 }
228:             }
229:             else if (strstr(line, KEYWORD_TQ)) {
230:                 // parse time_quantum
231:                 parse_tokens(two_tokens, line, &numTokens, "=");
232:                 if (numTokens == 2) {
233:                     // parse the second part using SPACE_CHARS
234:                     parse_tokens(queue_tokens, two_tokens[1], &n, SPACE_CHARS);
235:                     for (i = 0; i < n; i++)
236:                     {
237:                         sscanf(queue_tokens[i], "%d", &time_quantum[i]);
238:                     }
239:                 }
240:             }
```

```
241:             else if (strstr(line, KEYWORD_PROCESS_TABLE_SIZE)) {
242:                 // parse process_table_size
243:                 parse_tokens(two_tokens, line, &numTokens, equal_plus_spaces_delimiters);
244:                 if (numTokens == 2) {
245:                     sscanf(two_tokens[1], "%d", &process_table_size);
246:                 }
247:             }
248:             else if (strstr(line, KEYWORD_PROCESS_TABLE)) {
249:
250:                 // parse process_table
251:                 for (i=0; i<process_table_size; i++) {
252:
253:                     getline(&line, &len, fp);
254:                     line = strtok(line,"\n");
255:
256:                     sscanf(line, "%s %d %d", process_name, &process_arrival_time, &process
257:                     process_init(&process_table[i], process_name, process_arrival_time, pr
258:
259:                 }
260:             }
261:
262:         }
263:
264:     }
265: }
266: // Helper: Display the parsed values
267: void print_parsed_values() {
268:     printf("%s = %d\n", KEYWORD_QUEUE_NUMBER, queue_num);
269:     printf("%s = ", KEYWORD_TQ);
270:     for (int i=0; i<queue_num; i++)
271:         printf("%d ", time_quantum[i]);
272:     printf("\n");
273:     printf("%s = \n", KEYWORD_PROCESS_TABLE);
274:     process_table_print(process_table, process_table_size);
275: }
276:
277: //This function is used for debugging
278: void debug_print_transition(int process_id, int from_level, int to_level) {
279:     printf("Process P%d moved from Queue %d to Queue %d\n", process_id + 1, from_level, to
280: }
281:
282:
283: // TODO: Implementation of MLFQ algorithm
284: void mlfq() {
285:
286:     struct GanttChartItem chart[MAX_GANTT_CHART];
287:     int sz_chart = 0;
288:
289:     // TODO: Write your code here to implement MLFQ
290:     // Tips: A simple array is good enough to implement a queue
291:     int total_burst_time = 0;
292:     for (int i = 0; i < process_table_size; i++) {
293:         process_table[i].remain_time = process_table[i].burst_time;
294:         total_burst_time += process_table[i].burst_time;
295:     }
296:
297:     // Initialize queues
298:     struct Queue queues[queue_num];
299:     int current_quantum[queue_num];  // Track current time quantum for each queue
300:     for(int i = 0; i < queue_num; i++) {
```

```
301:         queue_init(&queues[i]);
302:         current_quantum[i] = 0;
303:     }
304: 
305:     int time = 0;
306:     int completed = 0;
307: 
308:     while (completed < total_burst_time) {
309:         // Check for new arrivals
310:         for (int i = 0; i < process_table_size; i++) {
311:             if (process_table[i].arrival_time == time && process_table[i].remain_time > 0)
312:                 queue_enqueue(&queues[0], i);
313:                 //printf("time %d: %s arrives and is added to queue 0\n", time, process_ta
314:             }
315:         }
316:         //printf("current_queue_count %d: \n", queues[0].count);
317: 
318:         // Find the highest priority non-empty queue
319:         int current_queue = -1;
320:         for (int i = 0; i < queue_num; i++) {
321:             if (!queue_is_empty(&queues[i])) {
322:                 current_queue = i;
323:                 break;
324:             }
325:         }
326:         //printf("current_queue %d: \n", current_queue);
327: 
328:         // If all queues are empty, handle idle time
329:         if (current_queue == -1) {
330:             gantt_chart_update(chart, &sz_chart, "idle", 1);
331:             //printf("time %d: CPU idle\n", time);
332:             time++;
333:             completed++;
334:             continue;
335:         }
336: 
337:         // Process execution
338:         int current_process = queue_peek(&queues[current_queue]);
339: 
340:         // Execute the process
341:         process_table[current_process].remain_time--;
342:         current_quantum[current_queue]++;
343:         completed++;
344: 
345:         gantt_chart_update(chart, &sz_chart, process_table[current_process].name, 1);
346:         //printf("time %d: executing %s in queue %d\n", time, process_table[current_proces
347: 
348:         // Check if process is completed
349:         if (process_table[current_process].remain_time == 0) {
350:             queue_dequeue(&queues[current_queue]);
351:             current_quantum[current_queue] = 0;
352:             //printf("time %d: %s completed\n", time + 1, process_table[current_process].n
353:         }
354:         // Check if time quantum expired
355:         else if (current_quantum[current_queue] == time_quantum[current_queue]) {
356:             if (current_queue < queue_num - 1) {
357:                 // Move to lower priority queue
358:                 int process = queue_peek(&queues[current_queue]);
359:                 queue_dequeue(&queues[current_queue]);
360:                 queue_enqueue(&queues[current_queue + 1], process);
```

```
361:                 //printf("time %d: %s moved from queue %d to queue %d\n", time + 1, proces
362:             }
363:             current_quantum[current_queue] = 0;
364:         }
365:         time++;
366:     }
367:     // At the end, display the final Gantt chart
368:     gantt_chart_print(chart, sz_chart);
369: 
370: }
371: 
372: 
373: int main() {
374:     parse_input();
375:     print_parsed_values();
376:     mlfq();
377:     return 0;
378: }
```