Escuela Superior de Informática
Universidad de Castilla-La Mancha

Programación Concurrente y de Tiempo Real
Modelo C01 – Paso de Mensajes

Prueba de Laboratorio [Solución]

Modelo C01 – Paso de Mensajes

APELLIDOS: _____
NOMBRE: _____ **GRUPO DE LABORATORIO**: _____

Indicaciones:

- No se permiten libros, apuntes ni teléfonos móviles.
- Cuando tenga una solución al ejercicio (compilación + ejecución) muéstrela al profesor.
- Debe anotar su solución por escrito en el espacio disponible en este cuestionario.

Calificación

Enunciado

Construya, utilizando ANSI C estándar, tres ejecutables que modelen el siguiente sistema. La simulación constará de un proceso *manager* que almacenará en un array una cadena de caracteres obtenida de la línea de órdenes y encargará su procesamiento a procesos de dos tipos: *processor* y *decoder*. El usuario ejecutará un proceso *manager* indicándole cuatro argumentos:

```
./exec/manager <encoded_data> <key> <n_processors> <n_subvectors>
```

Este proceso *manager* cargará un vector con los elementos de la cadena *encoded_data*, empleando el punto como separador de elementos. Este vector estará formado por números enteros que tendrán que tratar los *processors* y el proceso *decoder*. Esta cadena se dividirá en tantos subvectores como se indique en el cuarto argumento *n_subvectors*. En *n_processors* se indicará el número de procesos de tipo *processor* que se lanzarán para tratar los subvectores.

Los *processors* se encargarán de sumar el valor especificado en el parámetro *key* a cada elemento del vector. Así, el proceso *manager* se encargará de repartir el procesamiento del vector original en subvectores, asignando en cada momento un subvector a un *processor* que esté ocioso, hasta que se asignen todos los subvectores. En cada asignación, el proceso *manager* les indicará a los *processors* el fragmento del vector que deben procesar, junto con la clave (*key*) que indicó el usuario por línea de órdenes.

Cuando todos los subvectores hayan sido procesados, el *manager* llamará al **único proceso *decoder*** del sistema, el cual se encargará de decodificar cada número del vector de elementos (ya procesados por los *processors*) en un carácter ASCII. El *decoder* trabajará directamente con el vector completo. La correspondencia de traducción se resume en la siguiente tabla (la última fila de la tabla es la correspondencia del carácter con su código ASCII necesario para realizar la decodificación). Si el número entero a decodificar es mayor que 52, se traducirá siempre como un espacio en blanco.

Entero	**1**	**2**	**...**	**25**	**26**	**27**	**28**	**...**	**51**	**52**	**>52**
Traducción	**a**	**b**	**...**	**y**	**z**	**A**	**B**	**...**	**Y**	**Z**	**(Espacio Blanco)**
Código ASCII	*97*	*98*	*...*	*121*	*122*	*65*	*66*	*...*	*89*	*90*	*32*

Consideraciones

- No es obligatorio, aunque sí muy recomendable, incluir la comprobación de errores.
- Preste especial atención a lograr el máximo paralelismo posible en la solución.

Resolución

Utilice el código fuente suministrado a continuación como plantilla para resolver el ejercicio. Este código no debe ser modificado. Únicamente debe incorporar su código en las secciones indicadas.

A continuación se muestra una tabla con los buzones de mensajes utilizados.

Buzón	Uso
`MQ_RAW_DATA`	Usado por el *manager* para enviar subvectores
`MQ_PROCESSED_DATA`	Usado por los *processors* para enviar subvectores ya procesados (clave sumada)
`MQ_ENCODED_DATA`	Usado por el *manager* para enviar todo el vector codificado
`MQ_DECODED_DATA`	Usado por el *decoder* para enviar todo el vector descodificado

Test de Resultado Correcto

Una vez resuelto el ejercicio, si ejecuta el *manager* con los siguientes argumentos (4 subvectores, 2 procesadores, empleando como clave 4), se debe obtener el resultado indicado a continuación. Lógicamente, los PIDs asociados a los *processors*, y el orden de ejecución de los mismos, podrá ser diferente.

```
./exec/manager 45.1.8.8.56.0.11.10.1 4 2 4
```

```
[MANAGER] 2 PROCESSOR processes created.
[MANAGER] 1 DECODER processes created.

----- [MANAGER] Tasks sent -----

[PROCESSOR] 2850 | Start: 2 End: 3
[PROCESSOR] 2851 | Start: 0 End: 1
[PROCESSOR] 2851 | Start: 4 End: 5
[PROCESSOR] 2850 | Start: 6 End: 8

----- [MANAGER] Printing result -----
Decoded result: Well done

----- [MANAGER] Terminating running child processes -----
[MANAGER] Terminating PROCESSOR process [2850]...
[MANAGER] Terminating PROCESSOR process [2851]...

----- [MANAGER] Freeing resources -----
```

🛠 Complete el resultado obtenido de la ejecución con la siguiente lista de argumentos (`make solution`):

```
./exec/manager 19.5.4.-3.8.-9.10.9.62.5.4.62.-9.62.23.5.5.-6.62.26.5.-8 10 4 7
```

```
Mensaje decodificado:
---------------------

Resultado: Congrats on a Good Job
```

Esqueleto de Código Fuente

A continuación se muestra el esqueleto de código fuente para resolver el ejercicio.

Makefile

```
1    DIROBJ := obj/
2    DIREXE := exec/
3    DIRHEA := include/
4    DIRSRC := src/
5
6    CFLAGS := -I$(DIRHEA) -c -Wall -std=c99
7    LDLIBS := -lrt
8    CC := gcc
9
10   all : dirs manager processor decoder
11
12   dirs:
13      mkdir -p $(DIROBJ) $(DIREXE)
14
15   manager: $(DIROBJ)manager.o
16      $(CC) -o $(DIREXE)$@ $^ $(LDLIBS)
17   processor: $(DIROBJ)processor.o
18      $(CC) -o $(DIREXE)$@ $^ $(LDLIBS)
19   decoder: $(DIROBJ)decoder.o
20      $(CC) -o $(DIREXE)$@ $^ $(LDLIBS)
21
22   $(DIROBJ)%.o: $(DIRSRC)%.c
23      $(CC) $(CFLAGS) $^ -o $@
24
25   test:
26      ./exec/manager 45.1.8.8.56.0.11.10.1 4 2 4
27   solution:
28      ./exec/manager 19.5.4.-3.8.-9.10.9.62.5.4.62.-9.62.23.5.5.-6.62.26.5.-8 10 4 7
29
30   clean :
31      rm -rf *~ core $(DIROBJ) $(DIREXE) $(DIRHEA)*~ $(DIRSRC)*~
```

definitions.h

```
32   #define MQ_RAW_DATA       "/mq_raw_data"
33   #define MQ_PROCESSED_DATA "/mq_processed_data"
34   #define MQ_ENCODED_DATA   "/mq_encoded_data"
35   #define MQ_DECODED_DATA   "/mq_decoded_data"
36
37   #define PROCESSOR_CLASS   "PROCESSOR"
38   #define PROCESSOR_PATH    "./exec/processor"
39   #define DECODER_CLASS     "DECODER"
40   #define DECODER_PATH      "./exec/decoder"
41
42   #define MAX_ARRAY_SIZE 1024
43   #define NUM_DECODERS 1
44   #define SEPARATOR "."
45   #define TRUE 1
46   #define FALSE 0
47
48   /* Used in MQ_RAW_DATA and MQ_PROCESSED_DATA */
49   struct MsgProcessor_t {
50     char data[MAX_ARRAY_SIZE]; /* Data of the subvector to be processed */
51     int  index_start;          /* Start subvector index */
52     int  n_elements;           /* Number of elements in the subvector */
53     int  key;                  /* Key to carry out the 'processing' */
54   };
55
56   /* Used in MQ_ENCODED_DATA and MQ_DECODED_DATA */
57   struct MsgDecoder_t {
58     char data[MAX_ARRAY_SIZE]; /* Full vector to be decoded */
59     int  n_elements;           /* Number of elements to be decoded */
60   };
61
62   enum ProcessClass_t {PROCESSOR, DECODER};
63
64   struct TProcess_t {
65     enum ProcessClass_t class; /* PROCESSOR or DECODER */
66     pid_t pid;                 /* Process ID */
67     char *str_process_class;   /* String representation of the process class */
68   };
```

manager.c

```
69   #define _POSIX_SOURCE
70   #define _BSD_SOURCE
71
72   #include <errno.h>
73   #include <mqueue.h>
74   #include <signal.h>
75   #include <stdio.h>
76   #include <stdlib.h>
77   #include <string.h>
78   #include <sys/mman.h>
79   #include <sys/stat.h>
80   #include <sys/types.h>
81   #include <sys/wait.h>
82   #include <unistd.h>
83
84   #include <definitions.h>
85
86   /* Total number of processes */
87   int g_nProcesses;
88   /* 'Process table' (child processes) */
89   struct TProcess_t *g_process_table;
90
91   /* Process management */
92   void create_processes_by_class(enum ProcessClass_t class, int n_processes,
                                   int index_process_table);
93   pid_t create_single_process(const char *class, const char *path, const char *argv);
94   void get_str_process_info(enum ProcessClass_t class, char **path, char **str_process_class);
95   void init_process_table(int n_processors, int n_decoders);
96   void terminate_processes();
97   void wait_processes();
98
99   /* Message queue management */
100  void create_message_queue(const char *mq_name, mode_t mode, long mq_maxmsg, long mq_msgsize,
                              mqd_t *q_handler);
101  void close_message_queues(mqd_t q_handler_raw_data, mqd_t q_handler_processed_data,
                              mqd_t q_handler_encoded_data, mqd_t q_handler_decoded_data);
102
103  /* Task management */
104  void send_raw_data(int key, int n_subvectors, struct MsgProcessor_t *msg_task,
                        struct MsgDecoder_t *msg_result, mqd_t q_handler_raw_data);
105  void receive_encoded_data(int n_subvectors, struct MsgProcessor_t *msg_task,
                               struct MsgDecoder_t *msg_result, mqd_t q_handler_processed_data);
106  void decode(struct MsgDecoder_t *msg_result, mqd_t q_handler_encoded_data,
                 mqd_t q_handler_decoded_data);
107
108  /* Auxiliar functions */
109  void free_resources();
110  void generate_message_with_input_data(struct MsgDecoder_t *msg_result,
                                           char *encoded_input_data);
111  void install_signal_handler();
112  void parse_argv(int argc, char *argv[], char **p_encoded_input_data, int *key,
                     int *n_processors, int *n_subvectors);
113  void print_result(struct MsgDecoder_t *msg_result);
114  void signal_handler(int signo);
115
116  /******************** Main function ********************/
117
118  int main(int argc, char *argv[]) {
119    mqd_t q_handler_raw_data, q_handler_processed_data;
120    mqd_t q_handler_encoded_data, q_handler_decoded_data;
121    mode_t mode_creat_read_only = (O_RDONLY | O_CREAT);
122    mode_t mode_creat_write_only = (O_WRONLY | O_CREAT);
123    struct MsgProcessor_t msg_task;
124    struct MsgDecoder_t msg_result;
125
126    char *encoded_input_data;
127    int key, n_processors, n_subvectors;
128
129    /* Install signal handler and parse arguments*/
130    install_signal_handler();
131    parse_argv(argc, argv, &encoded_input_data, &key, &n_processors, &n_subvectors);
132
133    /* Init the process table*/
134    init_process_table(n_processors, NUM_DECODERS);
135
136    /* Create message queues */
137    create_message_queue(MQ_RAW_DATA, mode_creat_write_only, n_subvectors,
                          sizeof(struct MsgProcessor_t), &q_handler_raw_data);
138    create_message_queue(MQ_PROCESSED_DATA, mode_creat_read_only, n_subvectors,
                          sizeof(struct MsgProcessor_t), &q_handler_processed_data);
139    create_message_queue(MQ_ENCODED_DATA, mode_creat_write_only, 1,
                          sizeof(struct MsgDecoder_t), &q_handler_encoded_data);
140    create_message_queue(MQ_DECODED_DATA, mode_creat_read_only, 1,
                          sizeof(struct MsgDecoder_t), &q_handler_decoded_data);
141
142    /* Create processes */
143    create_processes_by_class(PROCESSOR, n_processors, 0);
144    create_processes_by_class(DECODER, NUM_DECODERS, n_processors);
```

```
145     /* Generate a message with the input data */
146     generate_message_with_input_data(&msg_result, encoded_input_data);
147
148     /* Manage tasks */
149     send_raw_data(key, n_subvectors, &msg_task, &msg_result, q_handler_raw_data);
150     receive_encoded_data(n_subvectors, &msg_task, &msg_result, q_handler_processed_data);
151     decode(&msg_result, q_handler_encoded_data, q_handler_decoded_data);
152
153     /* Wait for the decoder process */
154     wait_processes();
155
156     /* Print the decoded text */
157     print_result(&msg_result);
158
159     /* Free resources and terminate */
160     close_message_queues(q_handler_raw_data, q_handler_processed_data,
                            q_handler_encoded_data, q_handler_decoded_data);
161     terminate_processes();
162     free_resources();
163
164     return EXIT_SUCCESS;
165   }
166
167   /******************** Process Management ********************/
168
169   void create_processes_by_class(enum ProcessClass_t class, int n_processes,
                                    int index_process_table) {
170     char *path = NULL, *str_process_class = NULL;
171     int i;
172     pid_t pid;
173
174     get_str_process_info(class, &path, &str_process_class);
175
176     for (i = index_process_table; i < (index_process_table + n_processes); i++) {
177       pid = create_single_process(path, str_process_class, NULL);
178
179       g_process_table[i].class = class;
180       g_process_table[i].pid = pid;
181       g_process_table[i].str_process_class = str_process_class;
182     }
183
184     printf("[MANAGER] %d %s processes created.\n", n_processes, str_process_class);
185     sleep(1);
186   }
187
188   pid_t create_single_process(const char *path, const char *class, const char *argv) {
189     pid_t pid;
190
191     switch (pid = fork()) {
192     case -1 :
193       fprintf(stderr, "[MANAGER] Error creating %s process: %s.\n", class, strerror(errno));
194       terminate_processes();
195       free_resources();
196       exit(EXIT_FAILURE);
197     /* Child process */
198     case 0 :
199       if (execl(path, class, argv, NULL) == -1) {
200         fprintf(stderr, "[MANAGER] Error using execl() in %s process: %s.\n",
                    class, strerror(errno));
201         exit(EXIT_FAILURE);
202       }
203     }
204
205     /* Child PID */
206     return pid;
207   }
208
209   void get_str_process_info(enum ProcessClass_t class, char **path,
                               char **str_process_class) {
210     switch (class) {
211     case PROCESSOR:
212       *path = PROCESSOR_PATH;
213       *str_process_class = PROCESSOR_CLASS;
214       break;
215     case DECODER:
216       *path = DECODER_PATH;
217       *str_process_class = DECODER_CLASS;
218       break;
219     }
220   }
221
222   void init_process_table(int n_processors, int n_decoders) {
223     int i;
224
225     /* Number of processes to be created */
226     g_nProcesses = n_processors + n_decoders;
227     /* Allocate memory for the 'process table' */
228     g_process_table = malloc(g_nProcesses * sizeof(struct TProcess_t));
229
230     /* Init the 'process table' */
231     for (i = 0; i < g_nProcesses; i++) {
232       g_process_table[i].pid = 0;
233     }
234   }
```

```
235  void terminate_processes() {
236    int i;
237
238    printf("\n----- [MANAGER] Terminating running child processes ----- \n");
239    for (i = 0; i < g_nProcesses; i++) {
240      /* Child process alive */
241      if (g_process_table[i].pid != 0) {
242        printf("[MANAGER] Terminating %s process [%d]...\n",
                 g_process_table[i].str_process_class, g_process_table[i].pid);
243        if (kill(g_process_table[i].pid, SIGINT) == -1) {
244          fprintf(stderr, "[MANAGER] Error using kill() on process %d: %s.\n",
                   g_process_table[i].pid, strerror(errno));
245        }
246      }
247    }
248  }
249
250  void wait_processes() {
251    int i;
252    pid_t pid;
253
254    /* Wait for the termination of the DECODER process */
255    pid = wait(NULL);
256    for (i = 0; i < g_nProcesses; i++) {
257      if (pid == g_process_table[i].pid) {
258        /* Update the 'process table' */
259        g_process_table[i].pid = 0;
260        /* Child process found */
261        break;
262      }
263    }
264  }
265
266  /******************** Message queue management ********************/
267
268  void create_message_queue(const char *mq_name, mode_t mode, long mq_maxmsg, long mq_msgsize,
                             mqd_t *q_handler) {
269    struct mq_attr attr;
270
271    attr.mq_maxmsg = mq_maxmsg;
272    attr.mq_msgsize = mq_msgsize;
273    *q_handler = mq_open(mq_name, mode, S_IWUSR | S_IRUSR, &attr);
274  }
275
276  void close_message_queues(mqd_t q_handler_raw_data, mqd_t q_handler_processed_data,
                             mqd_t q_handler_encoded_data, mqd_t q_handler_decoded_data) {
277    mq_close(q_handler_raw_data);
278    mq_close(q_handler_processed_data);
279    mq_close(q_handler_encoded_data);
280    mq_close(q_handler_decoded_data);
281  }
282
283  /******************** Task management ********************/
284
285  void send_raw_data(int key, int n_subvectors, struct MsgProcessor_t *msg_task,
                      struct MsgDecoder_t *msg_result, mqd_t q_handler_raw_data) {
286    int i;
287
288    msg_task->key = key;
289    /* n_subvectors tasks to be sent */
290    for (i = 0; i < n_subvectors; i++) {
291      /* Set the subvector indexes */
292      msg_task->index_start = i * (msg_result->n_elements / n_subvectors);
293      msg_task->n_elements = msg_result->n_elements / n_subvectors;
294      /* Last task -> adjust the value of n_elements */
295      if (i == n_subvectors - 1) {
296        msg_task->n_elements = msg_result->n_elements - msg_task->index_start;
297      }
298      /* Beware! Copy only the data related to a single subvector */
299      memcpy(msg_task->data, &(msg_result->data[msg_task->index_start]),
               msg_task->n_elements * sizeof(char));
300      mq_send(q_handler_raw_data, (const char *)msg_task, sizeof(struct MsgProcessor_t), 0);
301    }
302
303    printf("\n----- [MANAGER] Tasks sent ----- \n\n");
304  }
305
306  void receive_encoded_data(int n_subvectors, struct MsgProcessor_t *msg_task,
                             struct MsgDecoder_t *msg_result, mqd_t q_handler_processed_data) {
307    int i;
308
309    /* n_subvectors tasks to be received */
310    for (i = 0; i < n_subvectors; i++) {
311      mq_receive(q_handler_processed_data, (char *)msg_task,
                  sizeof(struct MsgProcessor_t), NULL);
312      /* Beware! Copy only the data related to the processed subvector */
313      memcpy(&(msg_result->data[msg_task->index_start]), msg_task->data,
               msg_task->n_elements * sizeof(char));
314    }
315  }
```

```
316  void decode(struct MsgDecoder_t *msg_result, mqd_t q_handler_encoded_data,
                 mqd_t q_handler_decoded_data) {
317    /* Rendezvous with the DECODER process */
318    mq_send(q_handler_encoded_data, (const char *)msg_result, sizeof(struct MsgDecoder_t), 0);
319    mq_receive(q_handler_decoded_data, (char *)msg_result, sizeof(struct MsgDecoder_t), NULL);
320  }
321
322  /******************** Auxiliar functions ********************/
323
324  void free_resources() {
325    printf("\n----- [MANAGER] Freeing resources ----- \n");
326
327    /* Free the 'process table' memory */
328    free(g_process_table);
329
330    /* Remove message queues */
331    mq_unlink(MQ_RAW_DATA);
332    mq_unlink(MQ_PROCESSED_DATA);
333    mq_unlink(MQ_ENCODED_DATA);
334    mq_unlink(MQ_DECODED_DATA);
335  }
336
337  void generate_message_with_input_data(struct MsgDecoder_t *msg_result,
                                        char *encoded_input_data) {
338    int i = 0;
339    char *encoded_character;
340
341    msg_result->data[0] = atoi(strtok(encoded_input_data, SEPARATOR));
342    while ((encoded_character = strtok(NULL, SEPARATOR)) != NULL) {
343      msg_result->data[++i] = atoi(encoded_character);
344    }
345    msg_result->n_elements = ++i;
346  }
347
348  void install_signal_handler() {
349    if (signal(SIGINT, signal_handler) == SIG_ERR) {
350      fprintf(stderr, "[MANAGER] Error installing signal handler: %s.\n", strerror(errno));
351      exit(EXIT_FAILURE);
352    }
353  }
354
355  void parse_argv(int argc, char *argv[], char **p_encoded_input_data, int *key,
                  int *n_processors, int *n_subvectors) {
356    if (argc != 5) {
357      fprintf(stderr, "Synopsis: ./exec/manager <encoded_data> <key> <n_processors>
                <n_subvectors>.\n");
358      exit(EXIT_FAILURE);
359    }
360
361    *p_encoded_input_data = argv[1];
362    *key = atoi(argv[2]);
363    *n_processors = atoi(argv[3]);
364    *n_subvectors = atoi(argv[4]);
365  }
366
367  void print_result(struct MsgDecoder_t *msg_result) {
368    int i;
369
370    printf("\n----- [MANAGER] Printing result ----- \n");
371    printf("Decoded result: ");
372    for (i = 0; i < msg_result->n_elements; i++) {
373      putchar(msg_result->data[i]);
374    }
375    printf("\n");
376  }
377
378  void signal_handler(int signo) {
379    printf("\n[MANAGER] Program termination (Ctrl + C).\n");
380    terminate_processes();
381    free_resources();
382    exit(EXIT_SUCCESS);
383  }
```

processor.c

```
384  #include <fcntl.h>
385  #include <mqueue.h>
386  #include <stdio.h>
387  #include <stdlib.h>
388  #include <sys/stat.h>
389  #include <sys/types.h>
390  #include <unistd.h>
391
392  #include <definitions.h>
393
394  /* Message queue management */
395  void open_message_queue(const char *mq_name, mode_t mode, mqd_t *q_handler);
396
397  /* Task management */
398  void process_raw_data(mqd_t q_handler_raw_data, mqd_t q_handler_processed_data);
399
400  /******************** Main function ********************/
401
402  int main(int argc, char *argv[]) {
403    mqd_t q_handler_raw_data, q_handler_processed_data;
404    mode_t mode_read_only = O_RDONLY;
405    mode_t mode_write_only = O_WRONLY;
406
407    /* Open message queues */
408    open_message_queue(MQ_RAW_DATA, mode_read_only, &q_handler_raw_data);
409    open_message_queue(MQ_PROCESSED_DATA, mode_write_only, &q_handler_processed_data);
410
411    /* Task management */
412    while (TRUE) {
413      process_raw_data(q_handler_raw_data, q_handler_processed_data);
414    }
415
416    return EXIT_SUCCESS;
417  }
418
419  /******************** Message queue management ********************/
420
421  void open_message_queue(const char *mq_name, mode_t mode, mqd_t *q_handler) {
422    *q_handler = mq_open(mq_name, mode);
423  }
424
425  /******************** Task management ********************/
426
427  void process_raw_data(mqd_t q_handler_raw_data, mqd_t q_handler_processed_data) {
```

🛠 Incluya el código para procesar subvectores *(Longitud aprox. ≈ 9 Líneas de código)*

```
int i;
struct MsgProcessor_t msg_task;

mq_receive(q_handler_raw_data, (char *)&msg_task, sizeof(struct MsgProcessor_t), NULL);
/* Only process the data related to the subvector received */
for (i = 0; i < msg_task.n_elements; i++) {
  msg_task.data[i] += msg_task.key;
}
mq_send(q_handler_processed_data, (const char *)&msg_task, sizeof(struct MsgProcessor_t), 0);

printf("[PROCESSOR] %d | Start: %d End: %d\n", getpid(), msg_task.index_start,
       msg_task.index_start + msg_task.n_elements - 1);
/* Dont remove; simulates complexity */
sleep(1);
```

```
428
429  }
```

decoder.c

```
430  #include <fcntl.h>
431  #include <mqueue.h>
432  #include <stdio.h>
433  #include <stdlib.h>
434  #include <sys/stat.h>
435  #include <sys/types.h>
436  #include <unistd.h>
437
438  #include <definitions.h>
439
440  /* Message queue management */
441  void open_message_queue(const char *mq_name, mode_t mode, mqd_t *q_handler);
442
443  /* Task management */
444  void decode_data(mqd_t q_handler_encoded_data, mqd_t q_handler_decoded_data);
445
446  /* Auxiliar functions */
447  void decode_single_character(char *c);
448
449  /******************** Main function ********************/
450
451  int main(int argc, char *argv[]) {
452    mqd_t q_handler_encoded_data, q_handler_decoded_data;
453    mode_t mode_read_only = O_RDONLY;
454    mode_t mode_write_only = O_WRONLY;
455
456    /* Open message queues */
457    open_message_queue(MQ_ENCODED_DATA, mode_read_only, &q_handler_encoded_data);
458    open_message_queue(MQ_DECODED_DATA, mode_write_only, &q_handler_decoded_data);
459
460    /* Task management */
461    decode_data(q_handler_encoded_data, q_handler_decoded_data);
462
463    return EXIT_SUCCESS;
464  }
```

Incluya el resto de código del proceso *decoder (Longitud aprox. ≈ 20 Líneas de código)*

```
/******************** Message queue management ********************/

void open_message_queue(const char *mq_name, mode_t mode, mqd_t *q_handler) {
  *q_handler = mq_open(mq_name, mode);
}

/******************** Task management ********************/

void decode_data(mqd_t q_handler_encoded_data, mqd_t q_handler_decoded_data) {
  int i;
  struct MsgDecoder_t msg_result;

  mq_receive(q_handler_encoded_data, (char *)&msg_result, sizeof(struct MsgDecoder_t), NULL);
  /* Decode all the encoded data */
  for (i = 0; i < msg_result.n_elements; i++) {
    decode_single_character(&(msg_result.data[i]));
  }
  mq_send(q_handler_decoded_data, (const char *)&msg_result, sizeof(struct MsgDecoder_t), 0);

  /* Dont remove; simulates complexity */
  sleep(1);
}

/******************** Auxiliar functions ********************/

void decode_single_character(char *c) {
  if (*c <= 26)      *c += 96;  /* Lowercase */
  else if (*c <= 52) *c += 38;  /* Uppercase */
  else               *c = 32;   /* Blank     */
}
```