

# Lab activity

## Model C01 – Message Passing

LAST NAME: \_\_\_\_\_  
NAME: \_\_\_\_\_ LAB GROUP: \_\_\_\_\_

### Instructions:

- You cannot use books, notes nor mobile phones.
- When you have a working solution (compilation + execution), show it to the lecturer.
- You must write down the source code of your solution.

Grade

## Statement

Build, using ANSI C, a system composed of **three executables** that simulates the following behavior. The system will be handled by a *manager process* that will load a string obtained from the command-line arguments and will assign the string decoding to a set of *processor processes* and to a single *decoder process*. The user will execute a *manager* process with the following arguments:

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

The *manager* will load a vector that comprises the elements of the string named *encoded\_data*, using the dot character as a separator. This vector will be composed of integer numbers that the *processors* and the *decoder* will have to process. Such string will be divided into as many subvectors as the number set in the fourth argument named *n\_subvectors*. The argument named *n\_processors* will determine the number of *processors* that the *manager* will create.

The *processors* will add the value specified in the *key* argument to each element of the vector. Thus, the *manager* will be responsible for requesting the processing of the original vector in subvectors, assigning one subvector to an idle *processor* at a time, until all the subvectors are actually assigned. Each request will determine the part of the vector a *processor* must handle, together with the *key* established by the user in the command-line arguments.

When all the subvectors are processed, the *manager* will interact with the only one *decoder process*, which will decode each one of the numbers of the vector into an ASCII value. The decoder will directly work with the full vector. The decoding algorithm is summarized in the table below (the last row shows ASCII values). Note that if the value to be decoded is greater than 52, then it will be decoded as a blank.

Integer Value	1	2	...	25	26	27	28	...	51	52	>52
Decoding	a	b	...	y	z	A	B	...	Y	Z	(Blank)
ASCII Code	97	98	...	121	122	65	66	...	89	90	32

## Considerations

- You are advised to check errors when working with the message queues.
- Pay special attention to achieve the highest parallelism level.

## Resolution

Use the given source code to resolve the proposed exercise. This template code must not be modified. Include the required code in the indicated sections (frames).

Next, a table with the used messages queues is shown.

Message queue	Use
MQ_RAW_DATA	Used by the <i>manager</i> to send subvectors
MQ_PROCESSED_DATA	Used by the <i>processors</i> to send processed subvectors (key added)
MQ_ENCODED_DATA	Used by the <i>manager</i> to send the whole encoded vector
MQ_DECODED_DATA	Used by the <i>decoder</i> to send the whole decoded vector

## Test example

Once a executable file has been generated, if you execute the following command (4 *subvectors*, 2 *processors*, using 4 as *key*)

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

the obtained result should be as follows (the PIDs and the order of the requested tasks will differ):

```
[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 -----
```

✂ Write down the result obtained when executing the following command (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
```

```
----- [MANAGER] Printing result -----
```

Decoded result:

## Source code template

Next, you can study the source code provided as a template for you to solve the exercise. **You must only include the code required to complete the processor/decoder processes.**

### 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 {
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 {
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 {
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,
93                               int index_process_table);
94 pid_t create_single_process(const char *class, const char *path, const char *argv);
95 void get_str_process_info(enum ProcessClass t class, char **path, char **str_process_class);
96 void init_process_table(int n_processors, int n_decoders);
97 void terminate_processes();
98 void wait_processes();
99
100 /* Message queue management */
101 void create_message_queue(const char *mq_name, mode_t mode, long mq_maxmsg, long mq_msgsize,
102                           mqd_t *q_handler);
103 void close_message_queues(mqd_t q_handler_raw_data, mqd_t q_handler_processed_data,
104                           mqd_t q_handler_encoded_data, mqd_t q_handler_decoded_data);
105
106 /* Task management */
107 void send_raw_data(int key, int n_subvectors, struct MsgProcessor t *msg_task,
108                  struct MsgDecoder t *msg_result, mqd_t q_handler_raw_data);
109 void receive_encoded_data(int n_subvectors, struct MsgProcessor t *msg_task,
110                           struct MsgDecoder t *msg_result, mqd_t q_handler_processed_data);
111 void decode(struct MsgDecoder t *msg_result, mqd_t q_handler_encoded_data,
112             mqd_t q_handler_decoded_data);
113
114 /* Auxiliar functions */
115 void free_resources();
116 void generate_message_with_input_data(struct MsgDecoder t *msg_result,
117                                       char *encoded_input_data);
118 void install_signal_handler();
119 void parse_argv(int argc, char *argv[], char **p_encoded_input_data, int *key,
120                int *n_processors, int *n_subvectors);
121 void print_result(struct MsgDecoder t *msg_result);
122 void signal_handler(int signo);
123
124 /***** Main function *****/
125
126 int main(int argc, char *argv[]) {
127     mqd_t q_handler_raw_data, q_handler_processed_data;
128     mqd_t q_handler_encoded_data, q_handler_decoded_data;
129     mode_t mode_create_read_only = (O_RDONLY | O_CREAT);
130     mode_t mode_create_write_only = (O_WRONLY | O_CREAT);
131     struct MsgProcessor t msg_task;
132     struct MsgDecoder t msg_result;
133
134     char *encoded_input_data;
135     int key, n_processors, n_subvectors;
136
137     /* Install signal handler and parse arguments */
138     install_signal_handler();
139     parse_argv(argc, argv, &encoded_input_data, &key, &n_processors, &n_subvectors);
140
141     /* Init the process table */
142     init_process_table(n_processors, NUM_DECODERS);
143
144     /* Create message queues */
145     create_message_queue(MQ_RAW_DATA, mode_create_write_only, n_subvectors,
146                         sizeof(struct MsgProcessor t), &q_handler_raw_data);
147     create_message_queue(MQ_PROCESSED_DATA, mode_create_read_only, n_subvectors,
148                         sizeof(struct MsgProcessor t), &q_handler_processed_data);
149     create_message_queue(MQ_ENCODED_DATA, mode_create_write_only, 1,
150                         sizeof(struct MsgDecoder t), &q_handler_encoded_data);
151     create_message_queue(MQ_DECODED_DATA, mode_create_read_only, 1,
152                         sizeof(struct MsgDecoder t), &q_handler_decoded_data);
153
154     /* Create processes */
155     create_processes_by_class(PROCESSOR, n_processors, 0);
156     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,
161                      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,
170                              int index_process_table) {
171     char *path = NULL, *str_process_class = NULL;
172     int i;
173     pid_t pid;
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",
201                     class, strerror(errno));
202             exit(EXIT_FAILURE);
203         }
204     }
205     /* Child PID */
206     return pid;
207 }
208
209 void get_str_process_info(enum ProcessClass t_class, char **path,
210                          char **str_process_class) {
211     switch (class) {
212     case PROCESSOR:
213         *path = PROCESSOR_PATH;
214         *str_process_class = PROCESSOR_CLASS;
215         break;
216     case DECODER:
217         *path = DECODER_PATH;
218         *str_process_class = DECODER_CLASS;
219         break;
220     }
221 }
222
223 void init_process_table(int n_processors, int n_decoders) {
224     int i;
225
226     /* Number of processes to be created */
227     g_nProcesses = n_processors + n_decoders;
228     /* Allocate memory for the 'process table' */
229     g_process_table = malloc(g_nProcesses * sizeof(struct TProcess_t));
230
231     /* Init the 'process table' */
232     for (i = 0; i < g_nProcesses; i++) {
233         g_process_table[i].pid = 0;
234     }
235 }

```

```

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",
243                 g_process_table[i].str_process_class, g_process_table[i].pid);
244             if (kill(g_process_table[i].pid, SIGINT) == -1) {
245                 fprintf(stderr, "[MANAGER] Error using kill() on process %d: %s.\n",
246                     g_process_table[i].pid, strerror(errno));
247             }
248         }
249     }
250
251 void wait_processes() {
252     int i;
253     pid_t pid;
254
255     /* Wait for the termination of the DECODER process */
256     pid = wait(NULL);
257     for (i = 0; i < g_nProcesses; i++) {
258         if (pid == g_process_table[i].pid) {
259             /* Update the 'process table' */
260             g_process_table[i].pid = 0;
261             /* Child process found */
262             break;
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,
269                          mqd_t *q_handler) {
270     struct mq_attr attr;
271
272     attr.mq_maxmsg = mq_maxmsg;
273     attr.mq_msgsize = mq_msgsize;
274     *q_handler = mq_open(mq_name, mode, S_IWUSR | S_IRUSR, &attr);
275 }
276
277 void close_message_queues(mqd_t q_handler_raw_data, mqd_t q_handler_processed_data,
278                          mqd_t q_handler_encoded_data, mqd_t q_handler_decoded_data) {
279     mq_close(q_handler_raw_data);
280     mq_close(q_handler_processed_data);
281     mq_close(q_handler_encoded_data);
282     mq_close(q_handler_decoded_data);
283 }
284
285 /***** Task management *****/
286
287 void send_raw_data(int key, int n_subvectors, struct MsgProcessor_t *msg_task,
288                  struct MsgDecoder_t *msg_result, mqd_t q_handler_raw_data) {
289     int i;
290
291     msg_task->key = key;
292     /* n_subvectors tasks to be sent */
293     for (i = 0; i < n_subvectors; i++) {
294         /* Set the subvector indexes */
295         msg_task->index_start = i * (msg_result->n_elements / n_subvectors);
296         msg_task->n_elements = msg_result->n_elements / n_subvectors;
297         /* Last task -> adjust the value of n_elements */
298         if (i == n_subvectors - 1) {
299             msg_task->n_elements = msg_result->n_elements - msg_task->index_start;
300         }
301         /* Beware! Copy only the data related to a single subvector */
302         memcpy(msg_task->data, &(msg_result->data[msg_task->index_start]),
303             msg_task->n_elements * sizeof(char));
304         mq_send(q_handler_raw_data, (const char *)msg_task, sizeof(struct MsgProcessor_t), 0);
305     }
306
307     printf("\n----- [MANAGER] Tasks sent ----- \n\n");
308 }
309
310 void receive_encoded_data(int n_subvectors, struct MsgProcessor_t *msg_task,
311                          struct MsgDecoder_t *msg_result, mqd_t q_handler_processed_data) {
312     int i;
313
314     /* n_subvectors tasks to be received */
315     for (i = 0; i < n_subvectors; i++) {
316         mq_receive(q_handler_processed_data, (char *)msg_task,
317             sizeof(struct MsgProcessor_t), NULL);
318         /* Beware! Copy only the data related to the processed subvector */
319         memcpy(&(msg_result->data[msg_task->index_start]), msg_task->data,
320             msg_task->n_elements * sizeof(char));
321     }
322 }

```

```

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) {

```



Include the code required to process subvectors (*Aprox. ≈ 9 lines*)

```

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 }

```

✂ Include the rest of the *decoder* process (Aprox. ≈ 20 lines)

