

# Lab activity [Solution]

## Model C02 – 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** (*manager*, *processor* and *counter*) that model a file processing system that allows to compute two values: the number of words that begin with a given pattern and ii) the number of digits included in the words that met the previous constraint. The user will execute a *manager* process with the following arguments:

```
./exec/manager <n_processors> <pattern> <file>
```

where *n\_processors* represents the number of processes of the *processor* class to be created, *pattern* represents the pattern to be matched at the beginning of the analyzed words, and *file* represents the path to the file to be processed.

The *manager* process will create *n\_processors* processors, which will receive each one of the text lines (one message per text line) that make up the file. For each received line, a *processor* will analyze its words to check if they begin with *pattern*. If so, the *processor* will send the word (comprised in a message) to the only one counter process, which will be responsible for counting the number of digits of such word. The *processor* will wait for the counter to send back the result (an integer value) before processing the next word.

The *manager* will receive the partial results computed by the processors (see structure *MsgResult\_t* in *definitions.h*) and, when it has them all, it will print the global result, showing the total number of words that begin with *pattern* and the total number of digits included in each one of them.

For example, executing `./exec/manager 4 Wh data/test.txt` will generate the following output:

```
[PROCESSOR 2548]: 'Wh' found in 'Wh1o+a17.' with 3 digits  
[PROCESSOR 2550]: 'Wh' found in 'Wha1lt22' with 4 digits  
... (more partial results here)
```

```
----- [MANAGER] Printing result -----  
9 words -- 23 digits
```

## 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_LINES	Used by the <i>manager</i> to send text lines
MQ_RESULTS	Used by the <i>processors</i> to send partial results to the <i>manager</i>
MQ_WORDS	Used by the <i>processors</i> to send words to the <i>counter</i>
MQ_NUMBER_DIGITS	Used by the <i>counter</i> to send the number of digits included in a word
MQ_MUTEX	Used by the <i>processors</i> to guarantee the exclusive use of the <i>counter</i>

## Test example

Once a executable file has been generated, if you execute the following command (`make test`)

```
./exec/manager 4 Wh data/test.txt
```

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

```
[MANAGER] 4 PROCESSOR processes created.
[MANAGER] 1 COUNTER processes created.
[PROCESSOR 2618]: 'Wh' found in 'Whallt22' with 4 digits
[PROCESSOR 2616]: 'Wh' found in 'Whlo+a17.' with 3 digits
[PROCESSOR 2619]: 'Wh' found in 'What0-+' with 1 digits
[PROCESSOR 2616]: 'Wh' found in 'Whalt2' with 2 digits
[PROCESSOR 2616]: 'Wh' found in 'What101' with 3 digits
[PROCESSOR 2617]: 'Wh' found in 'Whoa.' with 0 digits
[PROCESSOR 2618]: 'Wh' found in 'Whallt22' with 4 digits
[PROCESSOR 2617]: 'Wh' found in 'Whallt2' with 3 digits
[PROCESSOR 2617]: 'Wh' found in 'What101' with 3 digits
```

```
----- [MANAGER] Printing result -----
9 words -- 23 digits
```

✂ Write down the result obtained when executing the following command (`make solution`):

```
./exec/manager 5 aux data/test_resolucion.txt
```

**Obtained result:**


```
84 words -- 248 digits
```

## 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 empty frames.**

### 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 counter
11
12 dirs:
13     mkdir -p $(DIROBJ) $(DIREXE)
```

 Include the linking rules (6 Lines)

```
manager: $(DIROBJ)manager.o
    $(CC) -o $(DIREXE)$@ $^ $(LDLIBS)

processor: $(DIROBJ)processor.o
    $(CC) -o $(DIREXE)$@ $^ $(LDLIBS)

counter: $(DIROBJ)counter.o
    $(CC) -o $(DIREXE)$@ $^ $(LDLIBS)
```

```
14 $(DIROBJ)%.o: $(DIRSRC)%.c
15     $(CC) $(CFLAGS) $^ -o $@
16
17 test:
18     ./exec/manager 4 Wh data/test.txt
19
20 solution:
21     ./exec/manager 5 aux data/test_solution.txt
22
23 clean :
24     rm -rf *~ core $(DIROBJ) $(DIREXE) $(DIRHEA)*~ $(DIRSRC)*~
```

### definitions.h

```
25 #define MQ_LINES           "/mq_lines"
26 #define MQ_RESULTS        "/mq_results"
27 #define MQ_WORDS          "/mq_words"
28 #define MQ_NUMBER_DIGITS  "/mq_number_digits"
29 #define MQ_Mutex          "/mq_mutex"
30
31 #define PROCESSOR_CLASS   "PROCESSOR"
32 #define PROCESSOR_PATH    "./exec/processor"
33 #define COUNTER_CLASS     "COUNTER"
34 #define COUNTER_PATH      "./exec/counter"
35
36 #define MAX_LINE_SIZE 255
37 #define NUM_COUNTERS 1
38 #define WORD_SEPARATOR " "
39 #define TRUE 1
40 #define FALSE 0
41
42 /* Used in MQ_LINES */
43 struct MsgLine_t {
44     char line[MAX_LINE_SIZE];
45     char pattern[MAX_LINE_SIZE];
46 };
47
48 /* Used in MQ_RESULTS */
49 struct MsgResult_t {
50     int n_words;
51     int n_digits;
52 };
53
54 enum ProcessClass_t {PROCESSOR, COUNTER};
55
56 struct TProcess_t {
57     enum ProcessClass_t class; /* PROCESSOR or COUNTER */
58     pid_t pid; /* Process ID */
59     char *str_process_class; /* String representation of the process class */
60 };
```

## manager.c

```

61 #define POSIX_SOURCE
62 #define _BSD_SOURCE
63
64 #include <errno.h>
65 #include <mqueue.h>
66 #include <signal.h>
67 #include <stdio.h>
68 #include <stdlib.h>
69 #include <string.h>
70 #include <sys/mman.h>
71 #include <sys/stat.h>
72 #include <unistd.h>
73
74 #include <definitions.h>
75
76 /* Total number of processes */
77 int g_nProcesses;
78 /* 'Process table' (child processes) */
79 struct TProcess_t *g_process_table;
80
81 /* Process management */
82 void create_processes_by_class(enum ProcessClass_t class, int n_processes,
83                               int index_process_table);
84 pid_t create_single_process(const char *class, const char *path, const char *argv);
85 void get_str_process_info(enum ProcessClass_t class, char **path, char **str_process_class);
86 void init_process_table(int n_processors, int n_counters);
87 void terminate_processes();
88
89 /* Message queue management */
90 void create_message_queue(const char *mq_name, mode_t mode, long mq_maxmsg, long mq_msgsize,
91                           mqd_t *q_handler);
92 void close_message_queues(mqd_t q_handler_lines, mqd_t q_handler_results,
93                           mqd_t q_handler_words, mqd_t q_handler_number_digits,
94                           mqd_t q_handler_mutex);
95
96 /* Task management */
97 void send_lines(const char *filename, char *pattern, int *n_lines, mqd_t q_handler_lines);
98 void receive_partial_results(int n_lines, struct MsgResult_t *global_results,
99                              mqd_t q_handler_results);
100
101 /* Auxiliar functions */
102 void free_resources();
103 void install_signal_handler();
104 void parse_argv(int argc, char *argv[], int *n_processors,
105                char **p_pattern, char **p_filename);
106 void print_result(struct MsgResult_t *global_results);
107 void signal_handler(int signo);
108
109 /***** Main function *****/
110
111 int main(int argc, char *argv[]) {
112     mqd_t q_handler_lines, q_handler_results;
113     mqd_t q_handler_words, q_handler_number_digits, q_handler_mutex;
114     mode_t mode_creat_only = O_CREAT;
115     mode_t mode_creat_read_only = (O_RDONLY | O_CREAT);
116     mode_t mode_creat_write_only = (O_WRONLY | O_CREAT);
117     struct MsgResult_t global_results;
118     global_results.n_words = global_results.n_digits = 0;
119
120     char *pattern, *filename, token;
121     int n_processors, n_lines = 0;
122
123     /* Install signal handler and parse arguments */
124     install_signal_handler();
125     parse_argv(argc, argv, &n_processors, &pattern, &filename);
126
127     /* Init the process table */
128     init_process_table(n_processors, NUM_COUNTERS);
129
130     /* Create message queues */
131     create_message_queue(MQ_LINES, mode_creat_write_only, n_processors,
132                         sizeof(struct MsgLine_t), &q_handler_lines);
133     create_message_queue(MQ_RESULTS, mode_creat_read_only, n_processors,
134                         sizeof(struct MsgResult_t), &q_handler_results);
135     create_message_queue(MQ_WORDS, mode_creat_only, 1,
136                         MAX_LINE_SIZE * sizeof(char), &q_handler_words);
137     create_message_queue(MQ_NUMBER_DIGITS, mode_creat_only, 1,
138                         sizeof(int), &q_handler_number_digits);
139     create_message_queue(MQ_MUTEX, mode_creat_write_only, 1,
140                         sizeof(char), &q_handler_mutex);
141
142     /* Init the mutex mailbox */
143     mq_send(q_handler_mutex, &token, sizeof(char), 0);
144
145     /* Create processes */
146     create_processes_by_class(PROCESSOR, n_processors, 0);
147     create_processes_by_class(COUNTER, NUM_COUNTERS, n_processors);
148
149     /* Manage tasks */
150     send_lines(filename, pattern, &n_lines, q_handler_lines);
151     receive_partial_results(n_lines, &global_results, q_handler_results);

```

```

140  /* Print the decoded text */
141  print_result(&global_results);
142
143  /* Free resources and terminate */
144  close_message_queues(q_handler_lines, q_handler_results,
                      q_handler_words, q_handler_number_digits, q_handler_mutex);
145  terminate_processes();
146  free_resources();
147
148  return EXIT_SUCCESS;
149 }
150
151 /***** Process Management *****/
152
153 void create_processes_by_class(enum ProcessClass t_class, int n_processes,
                              int index_process_table) {
154     char *path = NULL, *str_process_class = NULL;
155     int i;
156     pid_t pid;
157
158     get_str_process_info(class, &path, &str_process_class);
159
160     for (i = index_process_table; i < (index_process_table + n_processes); i++) {
161         pid = create_single_process(path, str_process_class, NULL);
162
163         g_process_table[i].class = class;
164         g_process_table[i].pid = pid;
165         g_process_table[i].str_process_class = str_process_class;
166     }
167
168     printf("[MANAGER] %d %s processes created.\n", n_processes, str_process_class);
169     sleep(1);
170 }
171
172 pid_t create_single_process(const char *path, const char *class, const char *argv) {
173     pid_t pid;
174
175     switch (pid = fork()) {
176     case -1:
177         fprintf(stderr, "[MANAGER] Error creating %s process: %s.\n", class, strerror(errno));
178         terminate_processes();
179         free_resources();
180         exit(EXIT_FAILURE);
181         /* Child process */
182     case 0:
183         if (execl(path, class, argv, NULL) == -1) {
184             fprintf(stderr, "[MANAGER] Error using execl() in %s process: %s.\n",
                    class, strerror(errno));
185             exit(EXIT_FAILURE);
186         }
187     }
188
189     /* Child PID */
190     return pid;
191 }
192
193 void get_str_process_info(enum ProcessClass t_class,
                          char **path, char **str_process_class) {
194     switch (class) {
195     case PROCESSOR:
196         *path = PROCESSOR_PATH;
197         *str_process_class = PROCESSOR_CLASS;
198         break;
199     case COUNTER:
200         *path = COUNTER_PATH;
201         *str_process_class = COUNTER_CLASS;
202         break;
203     }
204 }
205
206 void init_process_table(int n_processors, int n_counters) {
207     int i;
208
209     /* Number of processes to be created */
210     g_nProcesses = n_processors + n_counters;
211     /* Allocate memory for the 'process table' */
212     g_process_table = malloc(g_nProcesses * sizeof(struct TProcess_t));
213
214     /* Init the 'process table' */
215     for (i = 0; i < g_nProcesses; i++) {
216         g_process_table[i].pid = 0;
217     }
218 }
219
220 void terminate_processes() {
221     int i;
222
223     printf("\n----- [MANAGER] Terminating running child processes ----- \n");
224     for (i = 0; i < g_nProcesses; i++) {
225         /* Child process alive */
226         if (g_process_table[i].pid != 0) {
227             printf("[MANAGER] Terminating %s process [%d]...\n",
                    g_process_table[i].str_process_class, g_process_table[i].pid);

```

```

228         if (kill(g_process_table[i].pid, SIGINT) == -1) {
229             fprintf(stderr, "[MANAGER] Error using kill() on process %d: %s.\n",
230                     g_process_table[i].pid, strerror(errno));
231         }
232     }
233 }
234
235 /***** Message queue management *****/
236
237 void create_message_queue(const char *mq_name, mode_t mode, long mq_maxmsg, long mq_msgsize,
                           mqd_t *q_handler) {

```

✂ Include the code of the *create\_message\_queue* function (Aprox ~ 4 lines)

```

struct mq_attr attr;

attr.mq_maxmsg = mq_maxmsg;
attr.mq_msgsize = mq_msgsize;
*q_handler = mq_open(mq_name, mode, S_IWUSR | S_IRUSR, &attr);

```

```

238 }
239
240 void close_message_queues(mqd_t q_handler_lines, mqd_t q_handler_results,
                           mqd_t q_handler_words, mqd_t q_handler_number_digits,
                           mqd_t q_handler_mutex) {
241     mq_close(q_handler_lines);
242     mq_close(q_handler_results);
243     mq_close(q_handler_words);
244     mq_close(q_handler_number_digits);
245     mq_close(q_handler_mutex);
246 }
247
248 /***** Task management *****/
249
250 void send_lines(const char *filename, char *pattern, int *n_lines, mqd_t q_handler_lines) {
251     FILE *fp;
252     char line[MAX_LINE_SIZE];
253     struct MsgLine_t msg_line;
254
255     /* Open the file */
256     if ((fp = fopen(filename, "r")) == NULL) {
257         fprintf(stderr, "\n----- [MANAGER] Error opening %s ----- \n\n", filename);
258         terminate_processes();
259         free_resources();
260         exit(EXIT_FAILURE);
261     }
262
263     /* Read one line at a time and send tasks */
264     while (fgets(line, sizeof(line), fp) != NULL) {
265         strcpy(msg_line.line, line);
266         strcpy(msg_line.pattern, pattern);
267         mq_send(q_handler_lines, (const char *)&msg_line, sizeof(struct MsgLine_t), 0);
268         ++*n_lines;
269     }
270     fclose(fp);
271 }
272
273 void receive_partial_results(int n_lines, struct MsgResult_t *global_results,
                              mqd_t q_handler_results) {

```

✂ Include the code of the *receive\_partial\_results* function (Aprox ~ 7 lines)

```

struct MsgResult_t partial_results;

while (n_lines > 0) {
    mq_receive(q_handler_results, (char *)&partial_results, sizeof(struct MsgResult_t), NULL);
    global_results->n_words += partial_results.n_words;
    global_results->n_digits += partial_results.n_digits;
    --n_lines;
}

```

```

275 }

```

```

276 /***** Auxiliar functions *****/
277
278 void free_resources() {
279     printf("\n----- [MANAGER] Freeing resources ----- \n");
280
281     /* Free the 'process table' memory */
282     free(g_process_table);
283
284     /* Remove message queues */
285     mq_unlink(MQ_LINES);
286     mq_unlink(MQ_RESULTS);
287     mq_unlink(MQ_WORDS);
288     mq_unlink(MQ_NUMBER_DIGITS);
289     mq_unlink(MQ_Mutex);
290 }
291
292 void install_signal_handler() {
293     if (signal(SIGINT, signal_handler) == SIG_ERR) {
294         fprintf(stderr, "[MANAGER] Error installing signal handler: %s.\n", strerror(errno));
295         exit(EXIT_FAILURE);
296     }
297 }
298
299 void parse_argv(int argc, char *argv[], int *n_processors,
300                 char **p_pattern, char **p_filename) {
301     if (argc != 4) {
302         fprintf(stderr, "Synopsis: ./exec/manager <n_processors> <pattern> <file>.\n");
303         exit(EXIT_FAILURE);
304     }
305     *n_processors = atoi(argv[1]);
306     *p_pattern = argv[2];
307     *p_filename = argv[3];
308 }
309
310 void print_result(struct MsgResult t *global_results) {
311     printf("\n----- [MANAGER] Printing result ----- \n");
312     printf("\t%d words -- %d digits\n", global_results->n_words, global_results->n_digits);
313 }
314
315 void signal_handler(int signo) {
316     printf("\n[MANAGER] Program termination (Ctrl + C).\n");
317     terminate_processes();
318     free_resources();
319     exit(EXIT_SUCCESS);
320 }

```

## processor.c

```

321 #include <errno.h>
322 #include <fcntl.h>
323 #include <mqueue.h>
324 #include <stdio.h>
325 #include <stdlib.h>
326 #include <string.h>
327 #include <sys/stat.h>
328 #include <sys/types.h>
329 #include <unistd.h>
330
331 #include <definitions.h>
332
333 /* Message queue management */
334 void open_message_queue(const char *mq_name, mode_t mode, mqd_t *q_handler);
335
336 /* Task management */
337 void process_line(struct MsgResult t *partial_results, mqd_t q_handler_lines,
338                  mqd_t q_handler_mutex, mqd_t q_handler_words,
339                  mqd_t q_handler_number_digits);
340 void send_partial_results(struct MsgResult t *partial_results, mqd_t q_handler_results);
341
342 /***** Main function *****/
343
344 int main(int argc, char *argv[]) {
345     mqd_t q_handler_lines, q_handler_results;
346     mqd_t q_handler_words, q_handler_number_digits, q_handler_mutex;
347     mode_t mode_read_only = O_RDONLY;
348     mode_t mode_write_only = O_WRONLY;
349     mode_t mode_read_write = O_RDWR;
350     struct MsgResult t partial_results;
351
352     /* Open message queues */
353     open_message_queue(MQ_LINES, mode_read_only, &q_handler_lines);
354     open_message_queue(MQ_RESULTS, mode_write_only, &q_handler_results);
355     open_message_queue(MQ_WORDS, mode_write_only, &q_handler_words);
356     open_message_queue(MQ_NUMBER_DIGITS, mode_read_only, &q_handler_number_digits);
357     open_message_queue(MQ_Mutex, mode_read_write, &q_handler_mutex);

```

```

356  /* Task management */
357  while (TRUE) {
358      process_line(&partial_results, q_handler_lines, q_handler_mutex,
                  q_handler_words, q_handler_number_digits);
359      send_partial_results(&partial_results, q_handler_results);
360  }
361  return EXIT_SUCCESS;
362  }
363
364  /***** Message queue management *****/
365
366  void open_message_queue(const char *mq_name, mode_t mode, mqd_t *q_handler) {
367      *q_handler = mq_open(mq_name, mode);
368  }
369
370  /***** Task management *****/
371
372  void process_line(struct MsgResult_t *partial_results, mqd_t q_handler_lines,
                  mqd_t q_handler_words, mqd_t q_handler_number_digits) {
373      char *word, *pattern, word_copy[MAX_LINE_SIZE], token;
374      int n_words = 0, n_digits = 0;
375      struct MsgLine_t msg_line;
376
377      /* Initialize number of digits */
378      partial_results->n_digits = 0;
379
380      /* Wait for a new task */
381      mq_receive(q_handler_lines, (char *)&msg_line, sizeof(struct MsgLine_t), NULL);
382      pattern = msg_line.pattern;
383
384      /* Word processing */
385      word = strtok(msg_line.line, WORD_SEPARATOR);
386      while (word != NULL) {
387          /* Start with 'pattern'? */
388          if (strncmp(word, pattern, strlen(pattern)) == 0) {
389              n_words++;
390
391              /* Extra copy to avoid inconsistencies */
392              strcpy(word_copy, word);
393
394              /* Mutex to guarantee the exclusive use of the COUNTER */
395              mq_receive(q_handler_mutex, &token, sizeof(char), NULL);
396              /* Rendezvous with the counter process */
397              mq_send(q_handler_words, word_copy, sizeof(word_copy), 0);
398              mq_receive(q_handler_number_digits, (char *)&n_digits, sizeof(int), NULL);
399              mq_send(q_handler_mutex, &token, sizeof(char), 0);
400
401              /* Update the number of digits for the processed line */
402              partial_results->n_digits += n_digits;
403
404              printf("[PROCESSOR %d]: '%s' found in '%s' with %d digits\n",
                     getpid(), pattern, word, n_digits);
405          }
406          word = strtok(NULL, WORD_SEPARATOR);
407      }
408
409      /* Dont remove (simulates complexity) */
410      sleep(1);
411
412      /* Update the number of words */
413      partial_results->n_words = n_words;
414  }
415
416  void send_partial_results(struct MsgResult_t *partial_results, mqd_t q_handler_results) {
417      mq_send(q_handler_results, (const char *)partial_results, sizeof(struct MsgResult_t), 0);
418  }

```



## counter.c

✂ Include the code of the *counter* process (Aprox.  $\approx 42$  lines)

```
#include <ctype.h>
#include <fcntl.h>
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

#include <definitions.h>

/* Message queue management */
void open_message_queue(const char *mq_name, mode_t mode, mqd_t *q_handler);

/* Task management */
void count_digits(mqd_t q_handler_words, int *n_digits);
void send_count(mqd_t q_handler_number_digits, int *n_digits);

/***** Main function *****/

int main(int argc, char *argv[]) {
    mqd_t q_handler_words, q_handler_number_digits;
    int n_digits;
    mode_t mode_read_only = O_RDONLY;
    mode_t mode_write_only = O_WRONLY;

    /* Open message queues */
    open_message_queue(MQ_WORDS, mode_read_only, &q_handler_words);
    open_message_queue(MQ_NUMBER_DIGITS, mode_write_only, &q_handler_number_digits);

    /* Task management */
    while (TRUE) {
        count_digits(q_handler_words, &n_digits);
        send_count(q_handler_number_digits, &n_digits);
    }

    return EXIT_SUCCESS;
}

/***** 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 count_digits(mqd_t q_handler_words, int *n_digits) {
    char word[MAX_LINE_SIZE];
    int i = 0;

    /* Wait for a word to count digits */
    mq_receive(q_handler_words, word, sizeof(word), NULL);

    *n_digits = 0;
    while (word[i]) {
        if (isdigit(word[i])) {
            ++*n_digits;
        }
    }
}

void send_count(mqd_t q_handler_number_digits, int *n_digits) {
    mq_send(q_handler_number_digits, (const char *)n_digits, sizeof(int), 0);
}
```