

Prueba de Laboratorio [Solución]

Modelo B01 – Semáforos y Memoria Compartida

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 cargará una cadena de la línea de órdenes y encargará su traducción a un conjunto de **procesos decoder**. El usuario ejecutará un proceso *manager* indicándole tres argumentos:

```
./exec/manager <cadena> <num_proc_decoder> <tam_max_tarea>
```

Este proceso *manager* cargará *cadena* en un array de caracteres empleando el punto como separador de elementos. Esta *cadena* estará formada por números enteros (entre 1 y 57) que tendrán que traducir a caracteres los procesos *decoder* (con ayuda del proceso *symbol_decoder*). En *num_proc_decoder* se indicará el número de procesos *decoder* que se lanzarán para traducir el array y *tam_max_tarea* será el tamaño máximo de cada tarea o subvector (cada proceso *decoder* traducirá del array original suministrado por el manager un máximo de *n_tasks* tareas). 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 traducción):

Entero	1	2	...	25	26	27	28	...	51	52	53	54	55	56	57
Traducción	a	b	...	y	z	A	B	...	Y	Z	.	,	!	?	_
Código ASCII	97	98	...	121	122	65	66	...	89	90	46	44	33	63	95

El sistema contará con un **único proceso symbol_decoder** que se encargará de traducir los símbolos de puntuación que se corresponden con los valores enteros del 53 al 57 inclusive. Así, cuando un proceso *decoder* encuentre uno de estos valores, 'despertará' al proceso *symbol_decoder* que escribirá en una variable de memoria compartida el resultado de traducir ese signo de puntuación, volviendo a 'dormir' después de realizar su trabajo. El proceso *decoder* leerá ese valor y lo usará como resultado de la traducción final.

Los procesos *decoder* atenderán peticiones de traducción hasta que el *manager* les envíe la señal de terminación. En cada petición el proceso *manager* les indicará el índice de *inicio* y *fin* de cada tarea (subvector). Los procesos *decoder* actualizarán directamente el resultado en el array creado por el *manager*.

Por ejemplo, ante una ejecución como:

```
./exec/manager 34.5.12.12.15.55 2 2
```

el proceso *manager* lanzará dos procesos *decoder* que traducirán la cadena de entrada (3 tareas de tamaño 2). El resultado de la traducción será el siguiente array de salida en ASCII, con representación en cadena de caracteres "Hello!":

Array de entrada	34	5	12	12	15	55
Array de salida (ASCII)	72	101	108	108	111	33
Representación como caracteres	'H'	'e'	'l'	'l'	'o'	'!'

Resolución

Utilice el código fuente suministrado a continuación como plantilla para resolver el ejercicio. Este código no debe ser modificado (salvo la inicialización de los semáforos en el proceso *manager*). Únicamente debe incorporar su código en la sección indicada. No realice comprobación de errores en los parámetros. El proceso *manager* se encarga de enviar la señal de terminación a los procesos *decoder* y al proceso *symbol_decoder* cuando éstos han calculado el resultado final. Preste especial atención a lograr el máximo paralelismo posible en la solución.

✂ Indique a continuación el valor de inicialización de los semáforos (código en *manager.c*):

Línea Código	Semáforo	Uso	Valor inicial
344	SEM_TASK_READY	Nueva orden; despierta a un proceso <i>decoder</i>	0
345	SEM_TASK_READ	Indica al <i>manager</i> que la orden fue leída	0
346	SEM_TASK_PROCESSED	La traducción de una tarea fue realizada	0
348	SEM_MUTEX	Uso exclusivo del proceso <i>symbol_decoder</i>	1
349	SEM_SYMBOL_READY	Despierta al proceso <i>symbol_decoder</i>	0
350	SEM_SYMBOL_DECODED	Espera fin del proceso <i>symbol_decoder</i>	0

Test de Resultado Correcto

Una vez resuelto el ejercicio, si ejecuta el *manager* con los siguientes argumentos (*make test*),

```
./exec/manager 34.5.12.12.15.55 2 2
```

el resultado debe ser algo similar a (naturalmente cambiará el PID de los procesos *decoder* y probablemente el orden de atención de las peticiones):

```
./exec/manager 34.5.12.12.15.55 2 2
[PID 11997] Begin 0 End 1 Decoded Task He
[PID 11998] Begin 2 End 3 Decoded Task ll
[PID 11997] Begin 4 End 5 Decoded Task o!
Decoded result: Hello!
```

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

```
./exec/manager 35.57.18.5.1.12.12.25.57.12.9.11.5.57.20.8.9.19.57.20.5.19.20.55 5 4
```

Decoded result: **I_really_like_this_test!**

Esqueleto de Código Fuente

A continuación se muestra el esqueleto de código fuente para resolver el ejercicio. **Sólo debe modificar la inicialización de los semáforos e incluir la parte que falta del proceso *decoder* y el proceso *symbol_decoder*.**

Makefile

```

1  DIROBJ := obj/
2  DIREXE := exec/
3  DIRHEA := include/
4  DIRSRC := src/
5
6  CFLAGS := -I$(DIRHEA) -c -Wall -std=c99
7  LDLIBS := -pthread -lrt -lm
8  CC := gcc
9
10 all : dirs manager decoder symbol_decoder
11
12 dirs:
13     mkdir -p $(DIROBJ) $(DIREXE)
14
15 manager: $(DIROBJ)manager.o $(DIROBJ)semaphoreI.o
16     $(CC) -lm -o $(DIREXE)$@ $^ $(LDLIBS)
17
18 decoder: $(DIROBJ)decoder.o $(DIROBJ)semaphoreI.o
19     $(CC) -o $(DIREXE)$@ $^ $(LDLIBS)
20
21 symbol_decoder: $(DIROBJ)symbol_decoder.o $(DIROBJ)semaphoreI.o
22     $(CC) -o $(DIREXE)$@ $^ $(LDLIBS)
23
24 $(DIROBJ)%.o: $(DIRSRC)%.c
25     $(CC) $(CFLAGS) $^ -o $@
26
27 test:
28     ./exec/manager 34.5.12.12.15.55 2 2
29
30 solution:
31     ./exec/manager 35.57.18.5.1.12.12.25.57.12.9.11.5.57.20.8.9.19.57.20.5.19.20.55 5 4
32
33 clean :
34     rm -rf *~ core $(DIROBJ) $(DIREXE) $(DIRHEA)*~ $(DIRSRC)*~

```

definitions.h

```

35 #define SEM_TASK_READY      "sem_task_ready"
36 #define SEM_TASK_READ      "sem_task_read"
37 #define SEM_TASK_PROCESSED "sem_task_processed"
38 #define SHM_TASK           "shm_task"
39 #define SHM_DATA           "shm_data"
40
41 #define SEM_MUTEX           "sem_mutex"
42 #define SEM_SYMBOL_READY   "sem_symbol_ready"
43 #define SEM_SYMBOL_DECODED "sem_symbol_decoded"
44 #define SHM_SYMBOL         "shm_symbol"
45
46 #define DECODER_CLASS      "DECODER"
47 #define DECODER_PATH       "./exec/decoder"
48 #define SYMBOL_DECODER_CLASS "SYMBOL_DECODER"
49 #define SYMBOL_DECODER_PATH "./exec7symbol_decoder"
50
51 #define MAX_ARRAY_SIZE 1000
52 #define NUM_SYMBOL_DECODERS 1
53 #define SEPARATOR "."
54 #define TRUE 1
55 #define FALSE 0
56
57 struct TData_t {
58     char vector[MAX_ARRAY_SIZE];
59 };
60
61 struct TTask_t {
62     int begin;
63     int end;
64 };
65
66 struct TSymbol_t {
67     char value;
68 };
69
70 enum ProcessClass_t {DECODER, SYMBOL_DECODER};
71
72 struct TProcess_t {
73     enum ProcessClass_t class; /* DECODER or SYMBOL_DECODER */
74     pid_t pid; /* Process ID */
75     char *str_process_class; /* String representation of the process class */
76 };

```

semaphore.h

```
77 #ifndef __SEMAPHORE_H
78 #define __SEMAPHORE_H
79
80 #include <semaphore.h>
81
82 sem_t *create_semaphore (const char *name, unsigned int value);
83 sem_t *get_semaphore (const char *name);
84 void remove_semaphore (const char *name);
85 void signal_semaphore (sem_t *sem);
86 void wait_semaphore (sem_t *sem);
87
88 #endif
```

semaphore.c

```
89 #include <errno.h>
90 #include <fcntl.h>
91 #include <stdio.h>
92 #include <stdlib.h>
93 #include <string.h>
94 #include <unistd.h>
95
96 #include <semaphore.h>
97
98 sem_t *create_semaphore (const char *name, unsigned int value) {
99     sem_t *sem;
100
101     if ((sem = sem_open(name, O_CREAT, 0644, value)) == SEM_FAILED) {
102         fprintf(stderr, "Error creating semaphore <name>: %s\n", name, strerror(errno));
103         exit(EXIT_FAILURE);
104     }
105
106     return sem;
107 }
108
109 sem_t *get_semaphore (const char *name) {
110     sem_t *sem;
111
112     if ((sem = sem_open(name, O_RDWR)) == SEM_FAILED) {
113         fprintf(stderr, "Error retrieving semaphore <name>: %s\n", name, strerror(errno));
114         exit(EXIT_FAILURE);
115     }
116
117     return sem;
118 }
119
120 void remove_semaphore (const char *name) {
121     sem_t *sem = get_semaphore(name);
122
123     if ((sem_close(sem)) == -1) {
124         fprintf(stderr, "Error closing semaphore <name>: %s\n", name, strerror(errno));
125         exit(EXIT_FAILURE);
126     }
127
128     if ((sem_unlink(name)) == -1) {
129         fprintf(stderr, "Error removing semaphore <name>: %s\n", name, strerror(errno));
130         exit(EXIT_FAILURE);
131     }
132 }
133
134 void signal_semaphore (sem_t *sem) {
135     if ((sem_post(sem)) == -1) {
136         fprintf(stderr, "Error incrementing the semaphore: %s\n", strerror(errno));
137         exit(EXIT_FAILURE);
138     }
139 }
140
141 void wait_semaphore (sem_t *sem) {
142     if ((sem_wait(sem)) == -1) {
143         fprintf(stderr, "Error decrementing the semaphore: %s\n", strerror(errno));
144         exit(EXIT_FAILURE);
145     }
146 }
```

manager.c

```

147 #define POSIX_SOURCE
148 #define _BSD_SOURCE
149
150 #include <errno.h>
151 #include <fcntl.h>
152 #include <linux/limits.h>
153 #include <math.h>
154 #include <signal.h>
155 #include <stdio.h>
156 #include <stdlib.h>
157 #include <string.h>
158 #include <sys/mman.h>
159 #include <sys/stat.h>
160 #include <sys/types.h>
161 #include <unistd.h>
162
163 #include <definitions.h>
164 #include <semaphoreI.h>
165
166 /* Total number of processes */
167 int g_nProcesses;
168 /* 'Process table' (child processes) */
169 struct TProcess_t *g_process_table;
170
171 /* Process management */
172 void create_processes_by_class(enum ProcessClass t_class,
173                               int n_processes, int index_process_table);
174 pid_t create_single_process(const char *class, const char *path, const char *argv);
175 void get_str_process_info(enum ProcessClass t_class, char **path, char **str_process_class);
176 void init_process_table(int n_decoders, int n_symbol_decoders);
177 void terminate_processes();
178
179 /* Semaphores and shared memory management */
180 void create_shm_segments(int *shm_data, int *shm_task, int *shm_symbol,
181                          struct TData_t **p_data, struct TTask_t **p_task,
182                          char *encoded_input_data, int *n_input_data);
183 void create_sems(sem_t **p_sem_task_ready, sem_t **p_sem_task_read,
184                  sem_t **p_sem_task_processed);
185 void close_shared_memory_segments(int shm_data, int shm_task, int shm_symbol);
186
187 /* Task management */
188 void notify_tasks(sem_t *sem_task_ready, sem_t *sem_task_read, struct TTask_t *task,
189                  int max_task_size, int *n_tasks, int n_input_data);
190 void wait_tasks_termination(sem_t *sem_task_processed, int n_tasks);
191
192 /* Auxiliar functions */
193 void free_resources();
194 void install_signal_handler();
195 void parse_argv(int argc, char *argv[], char **p_encoded_input_data,
196                int *n_decoders, int *max_task_size);
197 void print_result(struct TData_t *data, int n_input_data);
198 void signal_handler(int signo);
199
200 /****** Main function *****/
201
202 int main(int argc, char *argv[]) {
203     struct TData_t *data;
204     struct TTask_t *task;
205     int shm_data, shm_task, shm_symbol;
206     sem_t *sem_task_ready, *sem_task_read, *sem_task_processed;
207
208     char *encoded_input_data;
209     int n_decoders, max_task_size, n_input_data, n_tasks;
210
211     /* Install signal handler and parse arguments */
212     install_signal_handler();
213     parse_argv(argc, argv, &encoded_input_data, &n_decoders, &max_task_size);
214
215     /* Init the process table */
216     init_process_table(n_decoders, NUM_SYMBOL_DECODERS);
217
218     /* Create shared memory segments and semaphores */
219     create_shm_segments(&shm_data, &shm_task, &shm_symbol,
220                        &data, &task, encoded_input_data, &n_input_data);
221     create_sems(&sem_task_ready, &sem_task_read, &sem_task_processed);
222
223     /* Create processes */
224     create_processes_by_class(DECODER, n_decoders, 0);
225     create_processes_by_class(SYMBOL_DECODER, NUM_SYMBOL_DECODERS, n_decoders);
226
227     /* Manage tasks */
228     notify_tasks(sem_task_ready, sem_task_read, task, max_task_size, &n_tasks, n_input_data);
229     wait_tasks_termination(sem_task_processed, n_tasks);
230
231     /* Print the decoded text */
232     print_result(data, n_input_data);
233 }

```

```

226  /* Free resources and terminate */
227  close shared memory segments(shm_data, shm_task, shm_symbol);
228  terminate_processes();
229  free_resources();
230
231  return EXIT_SUCCESS;
232 }
233
234 /***** Process Management *****/
235
236 void create_processes_by_class(enum ProcessClass t_class, int n_processes,
                               int index_process_table) {
237     char *path = NULL, *str_process_class = NULL;
238     int i;
239     pid_t pid;
240
241     get_str_process_info(class, &path, &str_process_class);
242
243     for (i = index_process_table; i < (index_process_table + n_processes); i++) {
244         pid = create_single_process(path, str_process_class, NULL);
245
246         g_process_table[i].class = class;
247         g_process_table[i].pid = pid;
248         g_process_table[i].str_process_class = str_process_class;
249     }
250
251     printf("[MANAGER] %d %s processes created.\n", n_processes, str_process_class);
252     sleep(1);
253 }
254
255 pid_t create_single_process(const char *path, const char *class, const char *argv) {
256     pid_t pid;
257
258     switch (pid = fork()) {
259     case -1:
260         fprintf(stderr, "[MANAGER] Error creating %s process: %s.\n", class, strerror(errno));
261         terminate_processes();
262         free_resources();
263         exit(EXIT_FAILURE);
264         /* Child process */
265     case 0:
266         if (execl(path, class, argv, NULL) == -1) {
267             fprintf(stderr, "[MANAGER] Error using execl() in %s process: %s.\n",
268                     class, strerror(errno));
269             exit(EXIT_FAILURE);
270         }
271     }
272     /* Child PID */
273     return pid;
274 }
275
276 void get_str_process_info(enum ProcessClass t_class,
                           char **path, char **str_process_class) {
277     switch (class) {
278     case DECODER:
279         *path = DECODER_PATH;
280         *str_process_class = DECODER_CLASS;
281         break;
282     case SYMBOL_DECODER:
283         *path = SYMBOL_DECODER_PATH;
284         *str_process_class = SYMBOL_DECODER_CLASS;
285         break;
286     }
287 }
288
289 void init_process_table(int n_decoders, int n_symbol_decoders) {
290     int i;
291
292     /* Number of processes to be created */
293     g_nProcesses = n_decoders + n_symbol_decoders;
294     /* Allocate memory for the 'process table' */
295     g_process_table = malloc(g_nProcesses * sizeof(struct TProcess_t));
296     /* Init the 'process table' */
297     for (i = 0; i < g_nProcesses; i++) {
298         g_process_table[i].pid = 0;
299     }
300 }
301
302 void terminate_processes() {
303     int i;
304     printf("\n----- [MANAGER] Terminating running child processes ----- \n");
305     for (i = 0; i < g_nProcesses; i++) {
306         /* Child process alive */
307         if (g_process_table[i].pid != 0) {
308             printf("[MANAGER] Terminating %s process [%d]...\n",
309                    g_process_table[i].str_process_class, g_process_table[i].pid);
310             if (kill(g_process_table[i].pid, SIGINT) == -1) {
311                 fprintf(stderr, "[MANAGER] Error using kill() on process %d: %s.\n",
312                         g_process_table[i].pid, strerror(errno));
313             }
314         }
315     }
316 }

```

```

315 /***** Semaphores and shared memory management *****/
316
317 void create_shm_segments(int *shm_data, int *shm_task, int *shm_symbol,
                          struct TData_t **p_data, struct TTask_t **p_task,
                          char *encoded_input_data, int *n_input_data) {
318     int i = 0;
319     char *encoded_character;
320
321     /* Create and initialize shared memory segments */
322     *shm_data = shm_open(SHM_DATA, O_CREAT | O_RDWR, 0644);
323     ftruncate(*shm_data, sizeof(struct TData_t));
324     *p_data = mmap(NULL, sizeof(struct TData_t), PROT_READ | PROT_WRITE, MAP_SHARED,
                    *shm_data, 0);
325
326     *shm_task = shm_open(SHM_TASK, O_CREAT | O_RDWR, 0644);
327     ftruncate(*shm_task, sizeof(struct TTask_t));
328     *p_task = mmap(NULL, sizeof(struct TTask_t), PROT_READ | PROT_WRITE, MAP_SHARED,
                    *shm_task, 0);
329
330     *shm_symbol = shm_open(SHM_SYMBOL, O_CREAT | O_RDWR, 0644);
331     ftruncate(*shm_symbol, sizeof(char));
332     /* No need to map shm_symbol since the manager process does not use it */
333
334     /* Load encoded data */
335     (*p_data)->vector[i++] = atoi(strtok(encoded_input_data, SEPARATOR));
336     while ((encoded_character = strtok(NULL, SEPARATOR)) != NULL) {
337         (*p_data)->vector[i++] = atoi(encoded_character);
338     }
339     *n_input_data = i;
340 }
341
342 void create_sems(sem_t **p_sem_task_ready, sem_t **p_sem_task_read,
                  sem_t **p_sem_task_processed) {
343     /* Create and initialize semaphores */
344     *p_sem_task_ready = create_semaphore(SEM_TASK_READY, 0);
345     *p_sem_task_read = create_semaphore(SEM_TASK_READ, 0);
346     *p_sem_task_processed = create_semaphore(SEM_TASK_PROCESSED, 0);
347     /* The manager process only initializes the test, but it does not use them */
348     create_semaphore(SEM_MUTEX, 1);
349     create_semaphore(SEM_SYMBOL_READY, 0);
350     create_semaphore(SEM_SYMBOL_DECODED, 0);
351 }
352
353 void close_shared_memory_segments(int shm_data, int shm_task, int shm_symbol) {
354     close(shm_data);
355     close(shm_task);
356     close(shm_symbol);
357 }
358
359 /***** Task management *****/
360
361 void notify_tasks(sem_t *sem_task_ready, sem_t *sem_task_read, struct TTask_t *task,
                  int max_task_size, int *n_tasks, int n_input_data) {
362     int current_task = 0;
363
364     /* Number of subvectors */
365     *n_tasks = ceil(n_input_data / (float)max_task_size);
366
367     while (current_task < *n_tasks) {
368         /* Compute the task info */
369         task->begin = current_task * max_task_size;
370         task->end = task->begin + max_task_size - 1;
371         if (task->end > (n_input_data - 1)) {
372             task->end = n_input_data - 1;
373         }
374         current_task++;
375
376         /* Task notification through rendezvous */
377         signal_semaphore(sem_task_ready);
378         wait_semaphore(sem_task_read);
379     }
380 }
381
382 void wait_tasks_termination(sem_t *sem_task_processed, int n_tasks) {
383     int n_tasks_processed = 0;
384
385     while (n_tasks_processed < n_tasks) {
386         wait_semaphore(sem_task_processed);
387         n_tasks_processed++;
388     }
389 }
390
391 /***** Auxiliar functions *****/
392
393 void free_resources() {
394     printf("\n----- [MANAGER] Freeing resources ----- \n");
395
396     /* Free the 'process table' memory */
397     free(g_process_table);
398
399     /* Semaphores */
400     remove_semaphore(SEM_TASK_READY);
401     remove_semaphore(SEM_TASK_READ);
402     remove_semaphore(SEM_TASK_PROCESSED);

```

```

403     remove_semaphore(SEM_MUTEX);
404     remove_semaphore(SEM_SYMBOL_READY);
405     remove_semaphore(SEM_SYMBOL_DECODED);
406
407     /* Shared memory segments */
408     shm_unlink(SHM_TASK);
409     shm_unlink(SHM_DATA);
410     shm_unlink(SHM_SYMBOL);
411 }
412
413 void install_signal_handler() {
414     if (signal(SIGINT, signal_handler) == SIG_ERR) {
415         fprintf(stderr, "[MANAGER] Error installing signal handler: %s.\n", strerror(errno));
416         exit(EXIT_FAILURE);
417     }
418 }
419
420 void parse_argv(int argc, char *argv[], char **p_encoded_input_data,
421                 int *n_decoders, int *max_task_size) {
422     if (argc != 4) {
423         fprintf(stderr, "Synopsis: ./exec/manager <data> <n_decoders> <max_task_size>.\n");
424         exit(EXIT_FAILURE);
425     }
426     *p_encoded_input_data = argv[1];
427     *n_decoders = atoi(argv[2]);
428     *max_task_size = atoi(argv[3]);
429 }
430
431 void print_result(struct TData_t *data, int n_input_data) {
432     int i;
433
434     printf("\n----- [MANAGER] Printing result ----- \n");
435     printf("Decoded result: ");
436     for (i = 0; i < n_input_data; i++) {
437         printf("%c", data->vector[i]);
438     }
439     printf("\n");
440 }
441
442 void signal_handler(int signo) {
443     printf("\n[MANAGER] Program termination (Ctrl + C).\n");
444     terminate_processes();
445     free_resources();
446     exit(EXIT_SUCCESS);
447 }

```

decoder.c

```

448 #include <fcntl.h>
449 #include <stdio.h>
450 #include <stdlib.h>
451 #include <sys/mman.h>
452 #include <sys/stat.h>
453 #include <sys/types.h>
454 #include <unistd.h>
455
456 #include <definitions.h>
457 #include <semaphoreI.h>
458
459 /* Semaphores and shared memory retrieval */
460 void get_shm_segments(int *shm_data, int *shm_task, int *shm_symbol,
461                       struct TData_t **p_data, struct TTask_t **p_task,
462                       struct TSymbol_t **p_symbol);
463
464 void get_sems(sem_t **p_sem_task_ready, sem_t **p_sem_task_read,
465               sem_t **p_sem_task_processed, sem_t **p_sem_mutex,
466               sem_t **p_sem_symbol_ready, sem_t **p_sem_symbol_decoded);
467
468 /* Task management */
469 void get_and_process_task(sem_t *sem_task_ready, sem_t *sem_task_read,
470                           sem_t *sem_mutex, sem_t *sem_symbol_ready,
471                           sem_t *sem_symbol_decoded,
472                           struct TData_t *data, const struct TTask_t *task,
473                           struct TSymbol_t *symbol);
474 void notify_task_completed(sem_t *sem_task_processed);
475
476 /***** Main function *****/
477
478 int main(int argc, char *argv[]) {
479     struct TData_t *data;
480     struct TTask_t *task;
481     struct TSymbol_t *symbol;
482     int shm_data, shm_task, shm_symbol;
483     sem_t *sem_task_ready, *sem_task_read, *sem_task_processed;
484     sem_t *sem_mutex, *sem_symbol_ready, *sem_symbol_decoded;
485
486     /* Get shared memory segments and semaphores */
487     get_shm_segments(&shm_data, &shm_task, &shm_symbol, &data, &task, &symbol);
488     get_sems(&sem_task_ready, &sem_task_read, &sem_task_processed,
489             &sem_mutex, &sem_symbol_ready, &sem_symbol_decoded);

```



```

481  /* Will work until killed by the manager */
482  while (TRUE) {
483      get_and_process_task(sem task_ready, sem task_read, sem mutex, sem_symbol_ready,
                           sem_symbol_decoded, data, task, symbol);
484      notify_task_completed(sem_task_processed);
485  }
486
487  return EXIT_SUCCESS;
488 }
489
490 /***** Semaphores and shared memory retrieval *****/
491
492 void get_shm_segments(int *shm_data, int *shm_task, int *shm_symbol,
                       struct TData_t **data, struct TTask_t **task,
                       struct TSymbol_t **symbol) {
493     *shm_data = shm_open(SHM_DATA, O_RDWR, 0644);
494     *data = mmap(NULL, sizeof(struct TData_t), PROT_READ | PROT_WRITE, MAP_SHARED,
                  *shm_data, 0);
495
496     *shm_task = shm_open(SHM_TASK, O_RDWR, 0644);
497     *task = mmap(NULL, sizeof(struct TTask_t), PROT_READ | PROT_WRITE, MAP_SHARED,
                  *shm_task, 0);
498
499     *shm_symbol = shm_open(SHM_SYMBOL, O_RDWR, 0644);
500     *symbol = mmap(NULL, sizeof(struct TSymbol_t), PROT_READ | PROT_WRITE, MAP_SHARED,
                    *shm_symbol, 0);
501 }
502
503 void get_sems(sem_t **p_sem_task_ready, sem_t **p_sem_task_read,
               sem_t **p_sem_task_processed, sem_t **p_sem_mutex,
               sem_t **p_sem_symbol_ready, sem_t **p_sem_symbol_decoded) {
504     *p_sem_task_ready = get_semaphore(SEM_TASK_READY);
505     *p_sem_task_read = get_semaphore(SEM_TASK_READ);
506     *p_sem_task_processed = get_semaphore(SEM_TASK_PROCESSED);
507     *p_sem_mutex = get_semaphore(SEM_MUTEX);
508     *p_sem_symbol_ready = get_semaphore(SEM_SYMBOL_READY);
509     *p_sem_symbol_decoded = get_semaphore(SEM_SYMBOL_DECODED);
510 }
511
512 /***** Task management *****/
513
514 void get_and_process_task(sem_t *sem_task_ready, sem_t *sem_task_read,
                           sem_t *sem_mutex, sem_t *sem_symbol_ready,
                           sem_t *sem_symbol_decoded,
                           struct TData_t *data, const struct TTask_t * task,
                           struct TSymbol_t *symbol) {

```

✂ Incluya aquí el código de `get_and_process_task` (Longitud aprox. ≈ 36 Líneas de código)

Muestre en pantalla información sobre el PID del proceso que ha calculado cada subvector, el inicio y el fin y el resultado parcial de traducción, como en el siguiente ejemplo: **[PID 276] Begin 0 End 1 Decoded Task Ho**

```

515     int i, task_begin, task_end;
516
517     /* Task notification through rendezvous */
518     wait_semaphore(sem_task_ready);
519     task_begin = task->begin;
520     task_end = task->end;
521     signal_semaphore(sem_task_read);
522
523     /* Task processing */
524     for (i = task_begin; i <= task_end; i++) {
525         if (data->vector[i] <= 26) {
526             data->vector[i] += 96;
527         }
528         else {
529             if (data->vector[i] <= 52) {
530                 data->vector[i] += 38;
531             }
532             /* Symbol (need to interact with the symbol decoder) */
533             else {
534                 /* Mutex to guarantee the exclusive use of symbol_decoder */
535                 wait_semaphore(sem_mutex);
536                 symbol->value = data->vector[i];
537                 signal_semaphore(sem_symbol_ready);
538                 wait_semaphore(sem_symbol_decoded);
539                 data->vector[i] = symbol->value;
540                 signal_semaphore(sem_mutex);
541             }
542         }
543     }
544
545     /* Result translated */
546     printf ("[PID %d] Begin %d End %d Decoded Task ", getpid(), task_begin, task_end);

```

```

547     for (i = task_begin; i <= task_end; i++) {
548         printf("%c", data->vector[i]);
549     }
550     printf ("\n");

551 }
552
553 void notify_task_completed(sem_t * sem_task_processed) {
554     signal_semaphore(sem_task_processed);
555 }


```

symbol_decoder.c

```

556 #include <fcntl.h>
557 #include <stdio.h>
558 #include <stdlib.h>
559 #include <sys/mman.h>
560 #include <sys/stat.h>
561 #include <sys/types.h>
562 #include <unistd.h>
563
564 #include <definitions.h>
565 #include <semaphoreI.h>
566
567 /* Semaphores and shared memory retrieval */
568 void get_shm_segments(int *shm_symbol, struct TSymbol_t **p_symbol);
569 void get_sems(sem_t **p_sem_symbol_ready, sem_t **p_sem_symbol_decoded);
570
571 /* Task management */
572 void get_and_process_task(sem_t *sem_symbol_ready, sem_t *sem_symbol_decoded,
                          struct TSymbol_t *symbol);

```

 Incluya aquí el código del proceso symbol_decoder (Longitud aprox. ≈ 30 Líneas de código)

```

573 /***** Main function *****/
574
575 int main(int argc, char *argv[]) {
576     struct TSymbol_t *symbol;
577     int shm_symbol;
578     sem_t *sem_symbol_ready, *sem_symbol_decoded;
579
580     /* Get shared memory segments and semaphores */
581     get_shm_segments(&shm_symbol, &symbol);
582     get_sems(&sem_symbol_ready, &sem_symbol_decoded);
583
584     /* Will work until killed by the manager */
585     while (TRUE) {
586         get_and_process_task(sem_symbol_ready, sem_symbol_decoded, symbol);
587     }
588
589     return EXIT_SUCCESS;
590 }
591
592 /***** Semaphores and shared memory retrieval *****/
593
594 void get_shm_segments(int *shm_symbol, struct TSymbol_t **p_symbol) {
595     *shm_symbol = shm_open(SHM_SYMBOL, O_RDWR, 0644);
596     *p_symbol = mmap(NULL, sizeof(struct TSymbol_t), PROT_READ | PROT_WRITE, MAP_SHARED,
597                     *shm_symbol, 0);
598 }
599
600 void get_sems(sem_t **p_sem_symbol_ready, sem_t **p_sem_symbol_decoded) {
601     *p_sem_symbol_ready = get_semaphore(SEM_SYMBOL_READY);
602     *p_sem_symbol_decoded = get_semaphore(SEM_SYMBOL_DECODED);
603 }
604
605 /***** Task management *****/
606
607 void get_and_process_task(sem_t *sem_symbol_ready, sem_t *sem_symbol_decoded,
                          struct TSymbol_t *symbol) {
608     wait_semaphore(sem_symbol_ready);
609     switch (symbol->value){
610         case 53: symbol->value = 46; break;
611         case 54: symbol->value = 44; break;
612         case 55: symbol->value = 33; break;
613         case 56: symbol->value = 63; break;
614         case 57: symbol->value = 95; break;
615     }
616     signal_semaphore(sem_symbol_decoded);
617 }

```