

Prueba de Laboratorio

Modelo B02 – 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, los ejecutables que modelen un sistema que calcule la **fracción canónica** (irreducible) a partir de la descomposición del numerador y denominador como producto de sus factores primos del modo que se detalla a continuación:

- Un **proceso manager** recibe el numerador y el denominador de una fracción. Éstos se deberán descomponer factorialmente y sus potencias deberán ser reducidas. Para ello, se lanzará una serie de **procesos factorer**. Habrá un proceso *factorer* para cada número primo, es decir, uno para el 2, otro para el 3, otro para el 5, etc. hasta N_PRIME_NUMBERS.
- Cada uno de estos procesos *factorer* se encarga de calcular cuántas veces el numerador y el denominador que el proceso *manager* ha recibido son divisibles entre el número primo que dicho proceso *factorer* debe procesar, y que le ha sido indicado por el proceso *manager*. A continuación, deberá eliminar aquél con menor exponente, dejando la diferencia al contrario. Por ejemplo, si el proceso *manager* recibe los números 28 y 42, entonces se tendrá que calcular:

$$\frac{28}{42} = \frac{2^2 * 7^1}{2^1 * 3^1 * 7^1} = \frac{2^1}{3^1}$$

Para ello, el proceso *factorer* encargado de procesar el nº primo 2 deberá calcular que puede dividir el numerador 2 veces y el denominador 1 vez, dejando en el numerador 2^1 ; el proceso *factorer* encargado de procesar el nº primo 3 deberá calcular que puede dividir el numerador 0 veces y el denominador 1 vez, dejando en el denominador 3^1 ; y el proceso *factorer* encargado de procesar el nº primo 7 deberá calcular que puede dividir el numerador 1 vez y el denominador 1 vez, eliminando por completo las potencias asociadas a éste.

Para facilitar la implementación, dispone de vectores para almacenar los exponentes de las potencias de los números primos en los que se descomponen el numerador y denominador (ver *struct TData_t*). Así, los valores ya simplificados para los datos del ejemplo anterior serán (para los 4 primeros números primos):

Numerador: $2^1 3^0 5^0 7^0 \rightarrow$ vector *numerator_exponent* [1, 0, 0, 0]

Denominador: $2^0 3^1 5^0 7^0 \rightarrow$ vector *denominator_exponent* [0, 1, 0, 0]

Consideraciones:

- No es obligatorio, aunque sí muy recomendable, realizar la comprobación de errores.
- Preste especial atención a lograr el máximo paralelismo posible en la solución.
- Utilice la primitiva *memcpy* en el proceso *factorer* para copiar, en una variable local, la orden recibida del proceso *manager*.

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.

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

Línea Código	Semáforo	Uso	Inicialización
371	SEM_TASK_READY	Nueva orden; despierta a un proceso <i>factorer</i>	
372	SEM_TASK_READ	Indica al <i>manager</i> que la orden fue leída	
373	SEM_TASK_PROCESSED	El proceso <i>factorer</i> terminó su trabajo	

Test de Resultado Correcto

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

```
./exec/manager 995742720 26078976
```

el resultados debe ser el siguiente:

```
[MANAGER] 101 FACTORER processes created.
```

```
Result: ( 2^2 3^1 5^1 7^1 )/( 11^1 )
```

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

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

```
./exec/manager 995742720 2935296
```

Resultado:

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 en los procesos *manager* y *factorer*.

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 factorer
11
12 dirs:
13     mkdir -p $(DIROBJ) $(DIREXE)
14
15 manager: $(DIROBJ)manager.o $(DIROBJ)semaphoreI.o
16     $(CC) -lm -o $(DIREXE)$@ $^ $(LDLIBS)
17 factorer: $(DIROBJ)factorer.o $(DIROBJ)semaphoreI.o
18     $(CC) -o $(DIREXE)$@ $^ $(LDLIBS)
19
20 $(DIROBJ)%.o: $(DIRSRC)%.c
21     $(CC) $(CFLAGS) $^ -o $@
22
23 test:
24     ./exec/manager 995742720 26078976
25 solution:
26     ./exec/manager 995742720 2935296
27
28 clean :
29     rm -rf *~ core $(DIROBJ) $(DIREXE) $(DIRHEA)*~ $(DIRSRC)*~

```

definitions.h

```

30 #define SEM_TASK_READY      "sem_task_ready"
31 #define SEM_TASK_READ      "sem_task_read"
32 #define SEM_TASK_PROCESSED "sem_task_processed"
33 #define SHM_TASK           "shm_task"
34 #define SHM_DATA           "shm_data"
35
36 #define FACTORER_CLASS      "FACTORER"
37 #define FACTORER_PATH      "./exec/factorer"
38
39 #define N_PRIME_NUMBERS     101
40
41 struct TData t {
42     /* Numerator/denominator to factorize */
43     int numerator;
44     int denominator;
45     /* Exponents for each prime number of the numerator */
46     int numerator_exponents[N_PRIME_NUMBERS];
47     /* Exponents for each prime number of the denominator */
48     int denominator_exponents[N_PRIME_NUMBERS];
49 };
50
51 struct TTask t {
52     /* Actual Value of the prime number*/
53     int prime_number;
54     /* Position of the prime number within the list */
55     int prime_number_position;
56 };
57
58 enum ProcessClass_t {FACTORER};
59
60 struct TProcess t {
61     enum ProcessClass_t class; /* FACTORER */
62     pid_t pid;                /* Process ID */
63     char *str_process_class;   /* String representation of the process class */
64 };

```

semaphore.h

```

65 #ifndef __SEMAPHOREI_H
66 #define __SEMAPHOREI_H
67
68 #include <semaphore.h>
69
70 sem_t *create_semaphore (const char *name, unsigned int value);
71 sem_t *get_semaphore (const char *name);
72 void remove_semaphore (const char *name);
73 void signal_semaphore (sem_t *sem);
74 void wait_semaphore (sem_t *sem);
75
76 #endif

```

semaphore.c

```

77 #include <stdio.h>
78 #include <errno.h>
79 #include <string.h>
80 #include <stdlib.h>
81 #include <unistd.h>
82 #include <fcntl.h>
83
84 #include <semaforoI.h>
85
86 sem_t *crear_sem (const char *name, unsigned int valor) {
87     sem_t *sem;
88
89     sem = sem_open(name, O_CREAT, 0644, valor);
90     if (sem == SEM_FAILED) {
91         fprintf(stderr, "Error al crear el sem. <%=>: %s\n",
92             name, strerror(errno));
93         exit(EXIT_FAILURE);
94     }
95
96     return sem;
97 }
98
99 sem_t *get_sem (const char *name) {
100     sem_t *sem;
101
102     sem = sem_open(name, O_RDWR);
103     if (sem == SEM_FAILED) {
104         fprintf(stderr, "Error al obtener el sem. <%=>: %s\n",
105             name, strerror(errno));
106         exit(EXIT_FAILURE);
107     }
108
109     return sem;
110 }
111
112 void destruir_sem (const char *name) {
113     sem_t *sem = get_sem(name);
114
115     /* Se cierra el sem */
116     if ((sem_close(sem)) == -1) {
117         fprintf(stderr, "Error al cerrar el sem. <%=>: %s\n",
118             name, strerror(errno));
119         exit(EXIT_FAILURE);
120     }
121
122     /* Se elimina el sem */
123     if ((sem_unlink(name)) == -1) {
124         fprintf(stderr, "Error al destruir el sem. <%=>: %s\n",
125             name, strerror(errno));
126         exit(EXIT_FAILURE);
127     }
128 }
129
130 void signal_sem (sem_t *sem) {
131     if ((sem_post(sem)) == -1) {
132         fprintf(stderr, "Error al modificar el sem.: %s\n",
133             strerror(errno));
134         exit(EXIT_FAILURE);
135     }
136 }
137
138 void wait_sem (sem_t *sem) {
139     if ((sem_wait(sem)) == -1) {
140         fprintf(stderr, "Error al modificar el sem. : %s\n",
141             strerror(errno));
142         exit(EXIT_FAILURE);
143     }
144 }

```

manager.c

```

145 #define POSIX_SOURCE
146 #define _BSD_SOURCE
147
148 #include <errno.h>
149 #include <fcntl.h>
150 #include <linux/limits.h>
151 #include <math.h>
152 #include <signal.h>
153 #include <stdio.h>
154 #include <stdlib.h>
155 #include <string.h>
156 #include <sys/mman.h>
157 #include <sys/stat.h>
158 #include <sys/types.h>
159 #include <sys/wait.h>
160 #include <unistd.h>
161
162 #include <definitions.h>
163 #include <semaphoreI.h>
164
165 /* Total number of processes */
166 int g_nProcesses;
167 /* 'Process table' (child processes) */
168 struct TProcess_t *g_process_table;
169
170 /* First n prime numbers */
171 int g_primes[] = {
172     2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,
173     71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157,
174     163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251,
175     257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353,
176     359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457,
177     461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547
178 };
179
180 /* Process management */
181 void create_processes_by_class(enum ProcessClass_t class, int n_processes,
182                               int index_process_table);
182 pid_t create_single_process(const char *class, const char *path, const char *argv);
183 void get_str_process_info(enum ProcessClass_t class, char **path, char **str_process_class);
184 void init_process_table(int n_factorers);
185 void terminate_processes();
186 void wait_processes();
187
188 /* Semaphores and shared memory management */
189 void create_shm_segments(int *shm_data, int *shm_task,
190                          struct TData_t **p_data, struct TTask_t **p_task,
191                          int numerator, int denominator, int n_prime_numbers);
192 void create_sems(sem_t **p_sem_task_ready, sem_t **p_sem_task_read,
193                  sem_t **p_sem_task_processed);
194 void close_shared_memory_segments(int shm_data, int shm_task);
195
196 /* Task management */
197 void notify_tasks(sem_t *sem_task_ready, sem_t *sem_task_read,
198                  struct TTask_t *task, int n_tasks);
199 void wait_tasks_termination(sem_t *sem_task_processed, int n_tasks);
200
201 /* Auxiliar functions */
202 void free_resources();
203 void install_signal_handler();
204 void parse_argv(int argc, char *argv[], int *numerator, int *denominator);
205 void print_result(struct TData_t *data);
206 void signal_handler(int signo);
207
208 /****** Main function *****/
209
210 int main(int argc, char *argv[]) {
211     struct TData_t *data;
212     struct TTask_t *task;
213     int shm_data, shm_task;
214     sem_t *sem_task_ready, *sem_task_read, *sem_task_processed;
215
216     int numerator, denominator;
217
218     /* Install signal handler and parse arguments */
219     install_signal_handler();
220     parse_argv(argc, argv, &numerator, &denominator);
221
222     /* Init the process table */
223     init_process_table(N_PRIME_NUMBERS);
224
225     /* Create shared memory segments and semaphores */
226     create_shm_segments(&shm_data, &shm_task, &data, &task,
227                        numerator, denominator, N_PRIME_NUMBERS);
228     create_sems(&sem_task_ready, &sem_task_read, &sem_task_processed);
229
230     /* Create processes */
231     create_processes_by_class(FACTORER, N_PRIME_NUMBERS, 0);

```

```

229  /* Manage tasks */
230  notify_tasks(sem_task_ready, sem_task_read, task, N_PRIME_NUMBERS);
231  wait_tasks_termination(sem_task_processed, N_PRIME_NUMBERS);
232
233  /* Wait for child processes */
234  wait_processes();
235
236  /* Print the obtained result */
237  print_result(data);
238
239  /* Free resources and terminate */
240  close_shared_memory_segments(shm_data, shm_task);
241  free_resources();
242
243  return EXIT_SUCCESS;
244 }
245
246 /***** Process Management *****/
247
248 void create_processes_by_class(enum ProcessClass t_class, int n_processes,
                               int index_process_table) {
249     char *path = NULL, *str_process_class = NULL;
250     int i;
251     pid_t pid;
252
253     get_str_process_info(class, &path, &str_process_class);
254
255     for (i = index_process_table; i < (index_process_table + n_processes); i++) {
256         pid = create_single_process(path, str_process_class, NULL);
257
258         g_process_table[i].class = class;
259         g_process_table[i].pid = pid;
260         g_process_table[i].str_process_class = str_process_class;
261     }
262
263     printf("[MANAGER] %d %s processes created.\n", n_processes, str_process_class);
264     sleep(1);
265 }
266
267 pid_t create_single_process(const char *path, const char *class, const char *argv) {
268     pid_t pid;
269
270     switch (pid = fork()) {
271     case -1 :
272         fprintf(stderr, "[MANAGER] Error creating %s process: %s.\n", class, strerror(errno));
273         terminate_processes();
274         free_resources();
275         exit(EXIT_FAILURE);
276         /* Child process */
277     case 0 :
278         if (execl(path, class, argv, NULL) == -1) {
279             fprintf(stderr, "[MANAGER] Error using execl() in %s process: %s.\n",
280                     class, strerror(errno));
281             exit(EXIT_FAILURE);
282         }
283     }
284     /* Child PID */
285     return pid;
286 }
287
288 void get_str_process_info(enum ProcessClass t_class, char **path,
                           char **str_process_class) {
289     switch (class) {
290     case FACTORER:
291         *path = FACTORER_PATH;
292         *str_process_class = FACTORER_CLASS;
293         break;
294     }
295 }
296
297 void init_process_table(int n_factorers) {
298     int i;
299
300     /* Number of processes to be created */
301     g_nProcesses = n_factorers;
302     /* Allocate memory for the 'process table' */
303     g_process_table = malloc(g_nProcesses * sizeof(struct TProcess_t));
304
305     /* Init the 'process table' */
306     for (i = 0; i < g_nProcesses; i++) {
307         g_process_table[i].pid = 0;
308     }
309 }

```

```

310 void terminate_processes() {
311     int i;
312
313     printf("\n----- [MANAGER] Terminating running child processes ----- \n");
314     for (i = 0; i < g_nProcesses; i++) {
315         /* Child process alive */
316         if (g_process_table[i].pid != 0) {
317             if (kill(g_process_table[i].pid, SIGINT) == -1) {
318                 fprintf(stderr, "[MANAGER] Error using kill() on process %d: %s.\n",
319                     g_process_table[i].pid, strerror(errno));
320             }
321         }
322     }
323
324 void wait_processes() {
325     int i, n_processes = g_nProcesses;
326     pid_t pid;
327
328     /* Wait for the termination of FACTORER processes */
329     while (n_processes > 0) {
330         /* Wait for any FACTORER process */
331         pid = wait(NULL);
332         for (i = 0; i < g_nProcesses; i++) {
333             if (pid == g_process_table[i].pid) {
334                 /* Update the 'process table' */
335                 g_process_table[i].pid = 0;
336                 n_processes--;
337                 /* Child process found */
338                 break;
339             }
340         }
341     }
342 }
343
344 /***** Semaphores and shared memory management *****/
345
346 void create_shm_segments(int *shm_data, int *shm_task,
347     struct TData t *p_data, struct TTask t *p_task,
348     int numerator, int denominator, int n_prime_numbers) {
349     int i;
350
351     /* Create and initialize shared memory segments */
352     *shm_data = shm_open(SHM_DATA, O_CREAT | O_RDWR, 0644);
353     ftruncate(*shm_data, sizeof(struct TData t));
354     *p_data = mmap(NULL, sizeof(struct TData t), PROT_READ | PROT_WRITE,
355         MAP_SHARED, *shm_data, 0);
356
357     *shm_task = shm_open(SHM_TASK, O_CREAT | O_RDWR, 0644);
358     ftruncate(*shm_task, sizeof(struct TTask t));
359     *p_task = mmap(NULL, sizeof(struct TTask t), PROT_READ | PROT_WRITE,
360         MAP_SHARED, *shm_task, 0);
361
362     /* SHM data initialization */
363     (*p_data)->numerator = numerator;
364     (*p_data)->denominator = denominator;
365     for (i = 0; i < n_prime_numbers; i++) {
366         (*p_data)->numerator_exponents[i] = 0;
367         (*p_data)->denominator_exponents[i] = 0;
368     }
369
370 void create_sems(sem_t **p_sem_task_ready, sem_t **p_sem_task_read,
371     sem_t **p_sem_task_processed) {
372     /* Create and initialize semaphores */
373     *p_sem_task_ready = create_semaphore(SEM_TASK_READY, /* SEM INIT VALUE */);
374     *p_sem_task_read = create_semaphore(SEM_TASK_READ, /* SEM INIT VALUE */);
375     *p_sem_task_processed = create_semaphore(SEM_TASK_PROCESSED, /* SEM INIT VALUE */);
376
377 void close_shared_memory_segments(int shm_data, int shm_task) {
378     close(shm_data);
379     close(shm_task);
380 }
381
382 /***** Task management *****/
383
384 void notify_tasks(sem_t *sem_task_ready, sem_t *sem_task_read,
385     struct TTask t *task, int n_tasks) {
386     int i;
387
388     for (i = 0; i < n_tasks; i++) {
389         task->prime_number = g_primes[i];
390         task->prime_number_position = i;
391         /* Task notification through rendezvous */
392         signal_semaphore(sem_task_ready);
393         wait_semaphore(sem_task_read);
394     }
395 }

```

```
395 void wait_tasks_termination(sem_t *sem_task_processed, int n_tasks) {
```

✂ Incluye el código de gestión de la finalización de tareas (*Longitud aprox. ≈ 5 líneas*)

```
396 }
397
398 /***** Auxiliar functions *****/
399
400 void free_resources() {
401     printf("\n----- [MANAGER] Freeing resources ----- \n");
402
403     /* Free the 'process table' memory */
404     free(g_process_table);
405
406     /* Semaphores */
407     remove_semaphore(SEM_TASK_READY);
408     remove_semaphore(SEM_TASK_READ);
409     remove_semaphore(SEM_TASK_PROCESSED);
410
411     /* Shared memory segments*/
412     shm_unlink(SHM_TASK);
413     shm_unlink(SHM_DATA);
414 }
415
416 void install_signal_handler() {
417     if (signal(SIGINT, signal_handler) == SIG_ERR) {
418         fprintf(stderr, "[MANAGER] Error installing signal handler: %s.\n", strerror(errno));
419         exit(EXIT_FAILURE);
420     }
421 }
422
423 void parse_argv(int argc, char *argv[], int *numerator, int *denominator) {
424     if (argc != 3) {
425         fprintf(stderr, "Synopsis: ./exec/manager <numerator> <denominator>.\n");
426         exit(EXIT_FAILURE);
427     }
428
429     *numerator = atoi(argv[1]);
430     *denominator = atoi(argv[2]);
431 }
432
433 void print_result(struct TData_t *data) {
434     int i, n_prime_numbers;
435
436     n_prime_numbers = sizeof(g_primes) / sizeof(g_primes[0]);
437
438     printf("\nResult: ( ");
439     for (i = 0; i < n_prime_numbers; i++) {
440         if (data->numerator_exponents[i] > 0) {
441             printf("%d^%d ", g_primes[i], data->numerator_exponents[i]);
442         }
443     }
444     printf(") / ( ");
445     for (i = 0; i < n_prime_numbers; i++) {
446         if (data->denominator_exponents[i] > 0) {
447             printf("%d^%d ", g_primes[i], data->denominator_exponents[i]);
448         }
449     }
450     printf("\n");
451 }
452
453 void signal_handler(int signo) {
454     printf("\n[MANAGER] Program termination (Ctrl + C).\n");
455     terminate_processes();
456     free_resources();
457     exit(EXIT_SUCCESS);
458 }
```


factorer.c

```

459 #include <fcntl.h>
460 #include <stdio.h>
461 #include <stdlib.h>
462 #include <string.h>
463 #include <sys/mman.h>
464 #include <sys/stat.h>
465 #include <sys/types.h>
466 #include <unistd.h>
467
468 #include <definitions.h>
469 #include <semaphoreI.h>
470
471 /* Semaphores and shared memory retrieval */
472 void close_shared_memory_segments(int shm_data, int shm_task);
473 void get_shm_segments(int *shm_data, int *shm_task, struct TData_t **p_data,
474                      struct TTask_t **p_task);
475
476 void get_sems(sem_t **p_sem_task_ready, sem_t **p_sem_task_read,
477              sem_t **p_sem_task_processed);
478
479 /* Task management */
480 void get_and_process_task(sem_t *sem_task_ready, sem_t *sem_task_read,
481                          struct TData_t *data, const struct TTask_t *task);
482 void notify_task_completed(sem_t *sem_task_processed);
483
484 /* Auxiliar functions */
485 int how_many_times_divisible(int number, int prime);
486
487 /****** Main function *****/
488
489 int main(int argc, char *argv[]) {
490     struct TData_t *data;
491     struct TTask_t *task;
492     int shm_data, shm_task;
493     sem_t *sem_task_ready, *sem_task_read, *sem_task_processed;
494
495     /* Get shared memory segments and semaphores */
496     get_shm_segments(&shm_data, &shm_task, &data, &task);
497     get_sems(&sem_task_ready, &sem_task_read, &sem_task_processed);
498
499     /* One single iteration */
500     get_and_process_task(sem_task_ready, sem_task_read, data, task);
501     notify_task_completed(sem_task_processed);
502
503     close_shared_memory_segments(shm_data, shm_task);
504
505     return EXIT_SUCCESS;
506 }
507
508 /****** Semaphores and shared memory retrieval *****/
509
510 void close_shared_memory_segments(int shm_data, int shm_task) {
511     close(shm_data);
512     close(shm_task);
513 }
514
515 void get_shm_segments(int *shm_data, int *shm_task,
516                      struct TData_t **data, struct TTask_t **task) {
517     *shm_data = shm_open(SHM_DATA, O_RDWR, 0644);
518     *data = mmap(NULL, sizeof(struct TData_t), PROT_READ | PROT_WRITE,
519                 MAP_SHARED, *shm_data, 0);
520
521     *shm_task = shm_open(SHM_TASK, O_RDWR, 0644);
522     *task = mmap(NULL, sizeof(struct TTask_t), PROT_READ | PROT_WRITE,
523                 MAP_SHARED, *shm_task, 0);
524 }
525
526 void get_sems(sem_t **p_sem_task_ready, sem_t **p_sem_task_read,
527              sem_t **p_sem_task_processed) {
528     *p_sem_task_ready = get_semaphore(SEM_TASK_READY);
529     *p_sem_task_read = get_semaphore(SEM_TASK_READ);
530     *p_sem_task_processed = get_semaphore(SEM_TASK_PROCESSED);
531 }

```

✂ Incluya el código de las funciones de gestión de tareas (*Longitud aprox. ≈ 21 Líneas*)

```
525  /***** Auxiliar functions *****/
526
527  int how_many_times_divisible(int number, int prime) {
528      int times;
529
530      for (times = 0; !(number % prime); times++, number = (number / prime));
531
532      return times;
533  }
```