Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №5

Студент Топорков Павел
Группа ИУ7-53Б
Дисциплина Операционные системы

Преподаватель: _____ Рязанова Н. Ю.
подпись, дата Фамилия, И.О.

Оценка _____

Москва — 2021 г.

Оглавление

1 Задание №1

1.1 Задание

Написать программу, реализующую задачу «Производство-потребление» по алгоритму Э. Дейкстры с тремя семафорами: двумя считающими и одним бинарным. В программе должно создаваться не менее 3х процессов -производителей и 3х процессов – потребителей. В программе надо обеспечить случайные задержки выполнения созданных процессов. В программе для взаимодействия производителей и потребителей буфер создается в разделяемом сегменте. Обратите внимание на то, чтобы не работать с одиночной переменной, а работать именно с буфером, состоящим их N ячеек по алгоритму. Производители в ячейки буфера записывают буквы алфавита по порядку. Потребители считывают символы из доступной ячейки. После считывания буквы из ячейки следующий потребитель может взять букву из следующей ячейки.

1.2 Код программы

Листинг 1.1: task1

```
0  #include <sys/ipc.h>
1  #include <sys/sem.h>
2  #include <sys/shm.h>
3  #include <sys/stat.h>
4  #include <sys/wait.h>
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <unistd.h>
9  #include <time.h>
10
11 #define OK 0
12 #define SHARED_ERR 1
13 #define SEM_ERR 2
14 #define FORK_ERR 3
15
16 #define COUNT_PRODUCER 5
17 #define COUNT_CONSUMERS 5
```

```
18 #define BUFFER_LEN 10
19 #define SEMBUF_LEN 2
20 #define COUNT_SYM 4
21
22 #define BIN_SEM 0
23 #define EMPT_SEM 1
24 #define FULL_SEM 2
25
26 #define P -1
27 #define V 1
28
29
30 struct sembuf producer_act[SEMBUF_LEN] = {
31     {EMPT_SEM, P, 0},
32     {BIN_SEM,  P, 0}
33 };
34
35 struct sembuf consumer_act[SEMBUF_LEN] = {
36     {FULL_SEM, P, 0},
37     {BIN_SEM,  P, 0}
38 };
39
40 struct sembuf producer_deact[SEMBUF_LEN] =  {
41     {BIN_SEM,  V, 0},
42     {FULL_SEM, V, 0}
43 };
44
45 struct sembuf consumer_deact[SEMBUF_LEN] =  {
46     {BIN_SEM,  V, 0},
47     {EMPT_SEM, V, 0}
48 };
49
50 char *buffer, *cur_pos_write, *cur_pos_read, *sym;
51
52
53 void producer(const int semid, const int consumerNum) {
54     for (char i = 0; i < COUNT_SYM; ++i) {
55         sleep(rand() % 2);
56
57         if (semop(semid, producer_act, 2) == -1) {
58             perror("Operation with semaphores is impossible.");
59             exit(SEM_ERR);
60         }
61
62         buffer[(*cur_pos_write) % BUFFER_LEN] = (*sym)++;
63         fprintf(stdout, "Producer %d >>>>> %c\n",
64                 consumerNum, buffer[((*cur_pos_write)++) % BUFFER_LEN]);
65
```

```
66          if (semop(semid, producer_deact, 2) == -1) {
67              perror("Operation with semaphores is impossible.");
68              exit(SEM_ERR);
69          }
70      }
71  }
72
73
74  void consumer(const int semid, const int consumerNum) {
75      for (int i = 0; i < COUNT_SYM; i++) {
76          sleep(rand() % 5);
77
78          if (semop(semid, consumer_act, 2) == -1) {
79              perror("Operation with semaphores is impossible");
80              exit(SEM_ERR);
81          }
82
83          fprintf(stdout, "Consumer %d <<<<< %c\n",
84                  consumerNum, buffer[((*cur_pos_read)++) % BUFFER_LEN]);
85
86          if (semop(semid, consumer_deact, 2) == -1) {
87              perror("Operation with semaphores is impossible");
88              exit(SEM_ERR);
89          }
90      }
91  }
92
93
94  void execute_consumer(const int id_sem) {
95      pid_t pid;
96
97      for (int i = 0; i < COUNT_CONSUMERS; ++i) {
98          pid = fork();
99          if (pid == -1) {
100             puts("Can not fork!");
101             exit(FORK_ERR);
102         } else if (pid == 0) {
103             consumer(id_sem, i + 1);
104             exit(OK);
105         }
106     }
107 }
108
109
110 void execute_producer(const int id_sem) {
111     pid_t pid;
112
113     for (int i = 0; i < COUNT_PRODUCER; ++i) {
```

```
114         pid = fork();
115         if (pid == -1) {
116             puts("Can not fork!");
117             exit(FORK_ERR);
118         } else if (pid == 0) {
119             producer(id_sem, i + 1);
120             exit(OK);
121         }
122     }
123 }
124
125 void wait_processes() {
126     int status;
127     pid_t data_pr;
128
129     for (size_t i = 0; i < COUNT_PRODUCER + COUNT_CONSUMERS; ++i) {
130         data_pr = wait(&status);
131
132         fprintf(stdout, "Child process has finished, pid = %d, status = %d
    \n",
133                 data_pr, status);
134
135         if (WIFEXITED(status)) {
136             fprintf(stdout, "Child process (%d) finished, code = %d.\n\n",
137                     data_pr, WEXITSTATUS(status));
138         } else if (WIFSIGNALED(status)) {
139             fprintf(stdout, "Child process (%d) finished, code = %d.\n\n",
140                     data_pr, WTERMSIG(status));
141         } else if (WIFSTOPPED(status)) {
142             fprintf(stdout, "Child process (%d) finished, code = %d.\n\n",
143                     data_pr, WSTOPSIG(status));
144         }
145     }
146 }
147
148
149 int main() {
150     int id_shared, id_sem;
151     pid_t pid = getpid();
152
153     fprintf(stdout, "Parent process pid = %d\n", pid);
154     id_shared = shmget(IPC_PRIVATE, (BUFFER_LEN + 3) * sizeof(char),
155                        IPC_CREAT | S_IRWXU | S_IRWXG | S_IRWXO);
156
157     if (id_shared == -1) {
158         perror("Unable to create a shared area.");
159         exit(SHARED_ERR);
160     }
```

```
161
162     buffer = shmat(id_shared, 0, 0);
163     if ((*buffer) == -1) {
164         perror("Can not attach memory");
165         exit(SHARED_ERR);
166     }
167
168     cur_pos_write = buffer;
169     cur_pos_read = buffer + sizeof(char);
170     sym = buffer + (sizeof(char) * 2);
171     buffer = buffer + (sizeof(char) * 3);
172
173     *sym = 'A';
174     *cur_pos_write = 0;
175     *cur_pos_read = 0;
176
177     id_sem = semget(IPC_PRIVATE, 3, IPC_CREAT | S_IRWXU | S_IRWXG |
    S_IRWXO);
178     if (id_sem == -1) {
179         perror("Unable to create a semaphore.");
180         exit(SEM_ERR);
181     }
182
183     if (semctl(id_sem, BIN_SEM, SETVAL, 1) == -1 ||
184         semctl(id_sem, EMPT_SEM, SETVAL, BUFFER_LEN) == -1 ||
185         semctl(id_sem, FULL_SEM, SETVAL, 0) == -1)
186     {
187         perror("Can not set values semaphors.");
188         exit(SEM_ERR);
189     }
190
191     execute_producer(id_sem);
192     execute_consumer(id_sem);
193     wait_processes();
194
195     fprintf(stdout, "%d\n", *cur_pos_write);
196     if (shmdt(cur_pos_write) == -1) {
197         perror("Can not detach shared memory");
198         exit(SHARED_ERR);
199     }
200
201     if (pid != 0) {
202         if (shmctl(id_shared, IPC_RMID, NULL) == -1) {
203             perror("Can not free memory!");
204             exit(SHARED_ERR);
205         }
206     }
207
```

```
208     exit(OK);
209 }
```

1.3 Демонстрация работы программы

Вывод программы:
Parent process pid = 14307
Producer 1 >>>>> A
Producer 3 >>>>>> B
Producer 2 >>>>> C
Producer 4 >>>>> D
Producer 5 >>>>> E
Producer 1 >>>>> F
Consumer 1 <<<<< A
Consumer 3 <<<<< B
Producer 2>>>>>G
Producer 5 >>>>> H
Producer 4 >>>>> I
Producer 3 >>>>>J
Consumer 5 <<<<< C
Consumer 4 <<<<< D
Consumer 2 <<<<< E
Producer 1 >>>>> K
Producer 1 >>>>> L
Producer 2 >>>>> M
Producer 4 >>>>> N
Producer 5 >>>>> O
Child process has finished, pid = 14318, status = 0
Child process (14318) finished, code = 0.
Consumer 1 <<<<< F
Producer 3 >>>>> P
Consumer 4 <<<<< G
Producer 2 >>>>> Q

Consumer 3 <<<<< H
Producer 5 >>>>>R
Consumer 5 <<<<< I
Producer 3 >>>>> S
Consumer 2 <<<<< J
Producer 4 >>>>> T
Child process has finished, pid = 14320, status = 0
Child process (14320) finished, code = 0.
Child process has finished, pid = 14322, status = 0
Child process (14322) finished, code = 0.
Child process has finished, pid = 14319, status = 0
Child process (14319) finished, code = 0.
Child process has finished, pid = 14321, status = 0
Child process (14321) finished, code = 0.
Consumer 1 <<<<< K
Consumer 4 <<<<< L
Consumer 3 <<<<< M
Consumer 2 <<<<< N
Consumer 5 <<<<< O
Consumer 1 <<<<< P
Consumer 3 <<<<< Q
Consumer 2 <<<<< R
Consumer 5 <<<<< S
Consumer 4 <<<<< T
Child process has finished, pid = 14325, status = 0
Child process (14325) finished, code = 0.
Child process has finished, pid = 14324, status = 0
Child process (14324) finished, code = 0.
Child process has finished, pid = 14327, status = 0
Child process (14327) finished, code = 0.
Child process has finished, pid = 14323, status = 0
Child process (14323) finished, code = 0.
Child process has finished, pid = 14326, status = 0
Child process (14326) finished, code = 0.

2 Задание №2

2.1 Задание

Написать программу, реализующую задачу «Читатели – писатели» по монитору Хоара с четырьмя функциями: Начать чтение, Закончить чтение, Начать запись, Закончить запись. В программе всеми процессами разделяется одно единственное значение в разделяемой памяти. Писатели ее только инкрементируют, читатели могут только читать значение. Писателей д.б. не меньше 3х, а читателей не меньше 5ти. Для реализации взаимоисключения используются семафоры.

2.2 Код программы

Листинг 2.1: task2

```
0  #include <sys/ipc.h>
1  #include <sys/sem.h>
2  #include <sys/shm.h>
3  #include <sys/stat.h>
4  #include <sys/wait.h>
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <time.h>
9  #include <signal.h>
10 #include <unistd.h>
11 #include <stdbool.h>
12
13 #define OK 0
14 #define SHARED_ERR 1
15 #define SEM_ERR 2
16 #define FORK_ERR 3
17
18 #define COUNT_WRITERS 5
19 #define COUNT_READERS 5
20
21 #define START_SEM_LEN 5
22 #define STOP_SEM_LEN 1
23
24 #define COUNT 3
```

```
25
26 #define SEM_WRITER_ACT 0
27 #define SEM_READER_ACT 1
28 #define SEM_WRITER_WAITING 2
29 #define SEM_READER_WAITING 3
30
31 #define P -1
32 #define Z 0
33 #define V 1
34
35 int *data = NULL;
36
37 struct sembuf start_write[START_SEM_LEN] = {
38     {SEM_WRITER_WAITING, V, 0},
39     {SEM_READER_ACT,     Z, 0},
40     {SEM_WRITER_ACT,     Z, 0},
41     {SEM_WRITER_ACT,     V, 0},
42     {SEM_WRITER_WAITING, P, 0}
43 };
44
45 struct sembuf stop_write[STOP_SEM_LEN] = {
46     {SEM_WRITER_ACT,     P, 0}
47 };
48
49 struct sembuf start_read[START_SEM_LEN] = {
50     {SEM_READER_WAITING, V, 0},
51     {SEM_WRITER_WAITING, Z, 0},
52     {SEM_WRITER_ACT,     Z, 0},
53     {SEM_READER_ACT,     V, 0},
54     {SEM_READER_WAITING, P, 0}
55 };
56
57 struct sembuf stop_read[STOP_SEM_LEN] = {
58     {SEM_READER_ACT,     P, 0}
59 };
60
61
62 void starting_write(const int id_sem) {
63     if (semop(id_sem, start_write, START_SEM_LEN) == -1) {
64         perror("Operation with semaphores is impossible.");
65         exit(SEM_ERR);
66     }
67 }
68
69 void stoping_writer(const int id_sem) {
70     if (semop(id_sem, stop_write, STOP_SEM_LEN) == -1) {
71         perror("Operation with semaphores is impossible.");
72         exit(SEM_ERR);
```

```
73      }
74  }
75
76  void run_writer(const int id_sem, const int id) {
77      for (int i = 0; i < COUNT; ++i) {
78          sleep(rand() % 4);
79          starting_write(id_sem);
80          (*data)++;
81          printf("Writer %d >>>> %d\n" , id, *data );
82          stoping_writer(id_sem);
83      }
84  }
85
86
87  void execute_writer(const int id_sem) {
88      pid_t pid;
89
90      for (int i = 0; i < COUNT_WRITERS; ++i) {
91          pid = fork();
92          if (pid == -1) {
93              puts("Can not fork!");
94              exit(FORK_ERR);
95          } else if (pid == 0) {
96              run_writer(id_sem, i);
97              exit(OK);
98          }
99      }
100 }
101
102
103 void starting_read(const int id_sem) {
104     if (semop(id_sem, start_read, START_SEM_LEN) == -1) {
105         perror("Operation with semaphores is impossible.");
106         exit(SEM_ERR);
107     }
108 }
109
110
111 void stoping_read(const int id_sem) {
112     if (semop(id_sem, stop_read, STOP_SEM_LEN) == -1) {
113         perror("Operation with semaphores is impossible.");
114         exit(SEM_ERR);
115     }
116 }
117
118 void run_read(const int id_sem, const int id) {
119     for (int i = 0; i < COUNT; ++i) {
120         sleep(rand() % 4);
```

```
121         starting_read(id_sem);
122         printf("Reader %d <<<< %d\n" , id, *data );
123         stoping_read(id_sem);
124     }
125 }
126
127
128 void execute_reader(const int id_sem) {
129     pid_t pid;
130
131     for (int i = 0; i < COUNT_WRITERS; ++i) {
132         pid = fork();
133         if (pid == -1) {
134             puts("Can not fork!");
135             exit(FORK_ERR);
136         } else if (pid == 0) {
137             run_read(id_sem, i);
138             exit(OK);
139         }
140     }
141 }
142
143 void wait_processes() {
144     int status;
145     pid_t data_pr;
146
147     for (size_t i = 0; i < COUNT_READERS + COUNT_WRITERS; ++i) {
148         data_pr = wait(&status);
149
150         fprintf(stdout, "Child process has finished, pid = %d, status = %d
    \n",
151                 data_pr, status);
152
153         if (WIFEXITED(status)) {
154             fprintf(stdout, "Child process (%d) finished, code = %d.\n\n",
155                     data_pr, WEXITSTATUS(status));
156         } else if (WIFSIGNALED(status)) {
157             fprintf(stdout, "Child process (%d) finished, code = %d.\n\n",
158                     data_pr, WTERMSIG(status));
159         } else if (WIFSTOPPED(status)) {
160             fprintf(stdout, "Child process (%d) finished, code = %d.\n\n",
161                     data_pr, WSTOPSIG(status));
162         }
163     }
164 }
165
166 int main() {
167     int id_shared, id_sem, status;
```

```
168     pid_t pid = getpid();
169
170     fprintf(stdout, "Parent process pid = %d\n", pid);
171     id_shared = shmget(IPC_PRIVATE, sizeof(int), IPC_CREAT | S_IRWXU |
    S_IRWXG | S_IRWXO);
172     if (id_shared == -1) {
173         perror("Unable to create a shared area.");
174         exit(SHARED_ERR);
175     }
176     data = shmat(id_shared, 0, 0);
177
178     id_sem = semget(IPC_PRIVATE, 5, IPC_CREAT | S_IRWXU | S_IRWXG |
    S_IRWXO);
179     if (id_sem == -1) {
180         perror("Unable to create a semaphore.");
181         exit(SEM_ERR);
182     }
183
184     if (semctl(id_sem, SEM_WRITER_ACT, SETVAL, 0) == -1 ||
185         semctl(id_sem, SEM_READER_ACT, SETVAL, 0) == -1 ||
186         semctl(id_sem, SEM_WRITER_WAITING, SETVAL, 0) == -1 ||
187         semctl(id_sem, SEM_READER_WAITING, SETVAL, 0) == -1)
188     {
189         perror("Can not set values semaphors.");
190         exit(SEM_ERR);
191     }
192
193     execute_writer(id_sem);
194     execute_reader(id_sem);
195     wait_processes();
196
197     if (shmdt(data) == -1) {
198         perror("Can not detach shared memory");
199         exit(SHARED_ERR);
200     }
201
202     if (pid != 0) {
203         if (shmctl(id_shared, IPC_RMID, NULL) == -1) {
204             perror("Can not free memory!");
205             exit(SHARED_ERR);
206         }
207     }
208
209     exit(OK);
210 }
```

2.3 Демонстрация работы программы

Вывод программы:

Parent process pid = 78761
Writer 0 >>>> 1
Reader 0 <<<< 1
Reader 2 <<<< 1
Reader 1 <<<< 1
Writer 1 >>>> 2
Writer 3 >>>> 3
Writer 4 >>>> 4
Writer 2 >>>> 5
Reader 4 <<<< 5
Reader 3 <<<< 5
Reader 0 <<<< 5
Reader 2 <<<< 5
Writer 3 >>>> 6
Writer 0 >>>> 7
Writer 1 >>>> 8
Reader 1 <<<< 8
Writer 4 >>>> 9
Writer 2 >>>> 10
Reader 3 <<<< 10
Reader 4 <<<< 10
Reader 2 <<<< 10
Writer 4 >>>> 11
Writer 2 >>>> 12
Reader 0 <<<< 12
Reader 3 <<<< 12
Reader 1 <<<< 12
Writer 3 >>>> 13
Writer 1 >>>> 14
Writer 0 >>>> 15
Reader 4 <<<< 15