|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **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 №** | 05 |

**Название:**

***Взаимодействие параллельных процессов***

**Дисциплина:  *Операционные системы***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент | ***ИУ7И-52Б*** |  |  | Чыонг Н. В. У. |
|  | (Группа) |  | (Подпись, дата) | (И.О. Фамилия) |
|  |  |  |  |  |
| Преподаватель |  |  |  | **Рязанова Н. Ю.** |
|  |  |  | (Подпись, дата) | (И.О. Фамилия) |

*Москва, 2021*

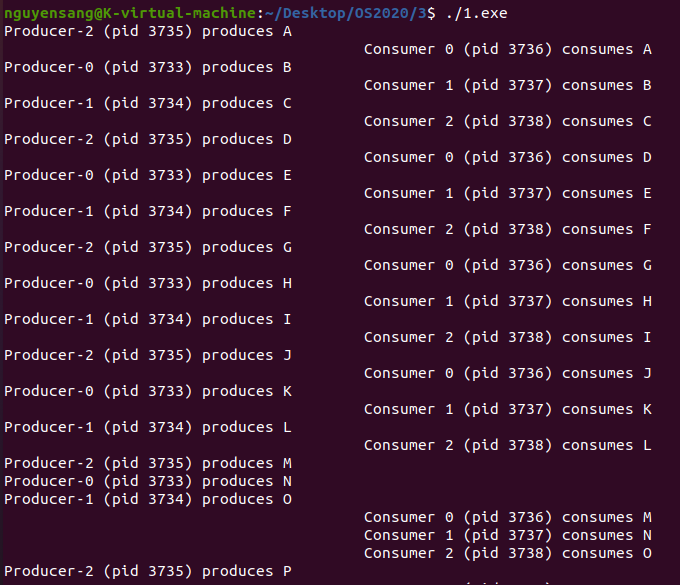
* **Задание 1:**

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. #include <signal.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <sys/stat.h>
5. #include <sys/sem.h>
6. #include <sys/shm.h>
7. #include <time.h>
8. #include <unistd.h>
9. #include <sys/wait.h>
11. #define BIN\_SEM   0
12. #define BUFFER\_EMPTY 1
13. #define BUFFER\_FULL  2
15. #define DEC -1
16. #define INC 1
18. #define PRODUCERS\_COUNT 3
19. #define CONSUMERS\_COUNT 3
21. #define PRODUCERS\_DELAY 3
22. #define CONSUMERS\_DELAY 2
24. **struct** sembuf producerP[2] = { {BUFFER\_EMPTY, DEC, SEM\_UNDO},
25. {BIN\_SEM, DEC, SEM\_UNDO}};
27. **struct** sembuf producerV[2] = { {BIN\_SEM, INC, SEM\_UNDO},
28. {BUFFER\_FULL, INC, SEM\_UNDO}};
30. **struct** sembuf consumerP[2] = { {BUFFER\_FULL,  DEC, SEM\_UNDO},
31. {BIN\_SEM,   DEC, SEM\_UNDO}};
33. **struct** sembuf consumerV[2] = { {BIN\_SEM, INC, SEM\_UNDO},
34. {BUFFER\_EMPTY, INC, SEM\_UNDO}};

37. #define N 26 /\*'Z' - 'A' \*/
39. #define PERMS S\_IRWXU | S\_IRWXG | S\_IRWXO
41. **int** semId = -1;
42. **int** shmId = -1;
44. **int** \*shm = NULL;
45. **int** \*shm\_producer\_count = NULL;
46. **int** \*shm\_consumer\_count = NULL;
47. **int** \*shm\_symbol\_now = NULL;
49. **int** randint(**int** a, **int** b)
50. {
51. **return** a + rand() % (b - a + 1);
52. }
54. **int** semrel(**int** semId)
55. {
56. **return** semctl(semId, 0, IPC\_RMID, 0);
57. }
59. **int** shmrel(**int** semId)
60. {
61. **return** shmctl(shmId, IPC\_RMID, NULL);
62. }
64. **void** forkChildren(**const** **int** n, **void** (\*func)(**const** **int**))
65. {
66. **for** (**int** i = 0; i < n; ++i)
67. {
68. **const** pid\_t pid = fork();
69. **if** (pid == -1)
70. {
71. perror("fork");
72. exit(1);
73. }
74. **else** **if** (pid == 0)
75. {
76. **if** (func)
77. func(i);
78. exit(1);
79. }
80. }
81. }
83. **void** waitChildren(**const** **int** n)
84. {
85. **for** (**int** i = 0; i < n; ++i)
86. {
87. **int** status;
88. **const** pid\_t child\_pid = wait(&status);
89. **if** (child\_pid == -1)
90. {
91. perror("wait");
92. exit(1);
93. }
94. **if** (WIFEXITED(status))
95. printf("Process %d returns %d\n", child\_pid, WEXITSTATUS(status));
96. **else** **if** (WIFSIGNALED(status))
97. printf("Process %d terminated with signal %d\n", child\_pid, WTERMSIG(status));
98. **else** **if** (WIFSTOPPED(status))
99. printf("Process %d stopped due signal %d\n", child\_pid, WSTOPSIG(status));
100. }
101. }
103. **void** producer(**const** **int** id)
104. {
105. **while**(1)
106. {
107. sleep(randint(0, PRODUCERS\_DELAY));
109. **if** (semop(semId, producerP, 2) == -1)
110. {
111. perror("semop");
112. exit(1);
113. }
115. /\* положить в буфер \*/
116. **int** symbol = 'A' + \*shm\_producer\_count % ('Z' - 'A');
117. \*(shm + \*shm\_producer\_count) = symbol;
118. printf("Producer-%d (pid %d) produces %c\n", id, getpid(), symbol);
119. (\*shm\_producer\_count)++;
121. **if** (semop(semId, producerV, 2) == -1)
122. {
123. perror("semop");
124. exit(1);
125. }
126. }
127. }
129. **void** consumer(**const** **int** id)
130. {
131. **while**(1)
132. {
133. sleep(randint(0, CONSUMERS\_DELAY));
135. **if** (semop(semId, consumerP, 2) == -1)
136. {
137. perror("semop");
138. exit(1);
139. }
140. /\* взять из буфера \*/
141. printf("\t\t\t\t\tConsumer %d (pid %d) consumes %c\n", id, getpid(), \*(shm + \*shm\_consumer\_count));
142. (\*shm\_consumer\_count)++;
144. **if** (semop(semId, consumerV, 2) == -1)
145. {
146. perror("semop");
147. exit(1);
148. }
149. }
150. }
152. **void** initSemaphore()
153. {
154. /\* два считающих семафора + один бинарный \*/
155. semId = semget(IPC\_PRIVATE, 3, IPC\_CREAT | PERMS);
157. **if** (semId == -1)
158. {
159. perror("semget");
160. exit(1);
161. }
162. /\*количество заполненных ячеек равно 0\*/
163. /\*Все ячейки буфера изначально пусты \*/
164. **if** (semctl(semId, BIN\_SEM,   SETVAL, 1) == -1 ||
165. semctl(semId, BUFFER\_EMPTY, SETVAL, N) == -1 ||
166. semctl(semId, BUFFER\_FULL,  SETVAL, 0) == -1)
167. {
168. perror("semctl");
169. exit(1);
170. }
171. }
173. **void** createSharedMemory()
174. {
175. // (N + 3) \* sizeof(int) - kích thước
176. //IPC\_PRIVATE - tạo seg mới
177. shmId = shmget(IPC\_PRIVATE, (N + 2) \* **sizeof**(**int**), IPC\_CREAT | PERMS);
178. **if** (shmId == -1)
179. {
180. perror("shmget");
181. exit(1);
182. }
183. shm = shmat(shmId, 0, 0);
184. **if** (shm == (**void** \*) -1)
185. {
186. perror("shmat");
187. exit(1);
188. }
189. shm\_producer\_count = shm;
190. shm\_consumer\_count = shm + 1;
191. \*shm\_producer\_count = 0;
192. \*shm\_consumer\_count = 0;
193. shm = shm + 2;
194. }
196. **int** main()
197. {
198. initSemaphore();
199. createSharedMemory();
201. forkChildren(PRODUCERS\_COUNT, producer);
202. forkChildren(CONSUMERS\_COUNT, consumer);
204. waitChildren(PRODUCERS\_COUNT + CONSUMERS\_COUNT);
206. shmrel(semId);
207. semrel(semId);
208. }



* **Задание 2:**

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. Для реализации взаимоисключения используются семафоры.

1. #include <sys/shm.h>
2. #include <sys/sem.h>
3. #include <fcntl.h>
4. #include <unistd.h>
5. #include <sys/wait.h>
6. #include <stdio.h>
7. #include <stdlib.h>
8. #include <signal.h>
9. #include <time.h>
10. #include <assert.h>
11. #include <sys/stat.h>
13. #define INC 1
14. #define DEC -1
15. #define CHK 0
17. #define PERMS S\_IRWXU | S\_IRWXG | S\_IRWXO
19. #define SEM\_ACTIVE\_WRITERS 0
20. #define SEM\_ACTIVE\_READERS 1
21. #define SEM\_WAITING\_WRITERS 2
22. #define SEM\_WAITING\_READERS 3
24. #define READER\_COUNT 5
25. #define WRITER\_COUNT 3
27. **struct** sembuf CANREAD   [3] = { {SEM\_ACTIVE\_WRITERS, CHK, 0},
28. {SEM\_WAITING\_WRITERS, CHK, 0},
29. {SEM\_WAITING\_READERS, INC, 0}};
31. **struct** sembuf STARTREAD [2] = { {SEM\_WAITING\_READERS, DEC, 0},
32. {SEM\_ACTIVE\_READERS, INC, 0}};
34. **struct** sembuf STOPREAD  [1] = { {SEM\_ACTIVE\_READERS, DEC, 0}};
36. **struct** sembuf CANWRITE  [3] = { {SEM\_ACTIVE\_READERS,  CHK, 0},
37. {SEM\_ACTIVE\_WRITERS,  CHK, 0},
38. {SEM\_WAITING\_WRITERS, INC, 0}};
40. **struct** sembuf STARTWRITE[2] = { {SEM\_ACTIVE\_WRITERS,  INC, 0},
41. {SEM\_WAITING\_WRITERS, DEC, 0}};
43. **struct** sembuf stopwrite [1] = { {SEM\_ACTIVE\_WRITERS, DEC, 0}};
45. #define CANREAD\_SIZE 3
46. #define STARTREAD\_SIZE 2
47. #define STOPREAD\_SIZE 1
48. #define CANWRITE\_SIZE 3
49. #define STARTWRITE\_SIZE 2
50. #define stopwrite\_SIZE 1
52. **int** sem\_id = -1;
53. **int** shm\_id = -1;
54. **int** \*shm = NULL;
56. **int** randint(**int** a, **int** b)
57. {
58. **return** a + rand() % (b - a + 1);
59. }
61. **int** initSemaphore()
62. {
63. sem\_id = semget(IPC\_PRIVATE, 5, IPC\_CREAT | PERMS);
64. **if** (sem\_id == -1)
65. {
66. perror("semget");
67. exit(1);
68. }
70. **if** (semctl(sem\_id, SEM\_ACTIVE\_WRITERS, SETVAL, 0) == -1 || semctl(sem\_id, SEM\_ACTIVE\_READERS, SETVAL, 0) == -1 ||
71. semctl(sem\_id, SEM\_WAITING\_READERS, SETVAL, 0) == -1 || semctl(sem\_id, SEM\_WAITING\_WRITERS, SETVAL, 0) == -1)
72. {
73. perror("semctl");
74. exit(1);
75. }
77. **return** sem\_id;
78. }
80. **void** forkChildren(**const** **int** n, **void** (\*func)(**const** **int**))
81. {
82. **for** (**int** i = 0; i < n; ++i)
83. {
84. **const** pid\_t pid = fork();
85. **if** (pid == -1)
86. {
87. perror("Err: fork");
88. exit(1);
89. }
90. **else** **if** (pid == 0)
91. {
92. **if** (func)
93. func(i);
94. exit(1);
95. }
96. }
97. }
99. **void** waitChildren(**const** **int** n)
100. {
101. **for** (**int** i = 0; i < n; ++i)
102. {
103. **int** status;
104. **const** pid\_t child\_pid = wait(&status);
105. **if** (child\_pid == -1)
106. {
107. perror("wait");
108. exit(1);
109. }
110. **if** (WIFEXITED(status))
111. printf("Process %d returns %d\n", child\_pid, WEXITSTATUS(status));
112. **else** **if** (WIFSIGNALED(status))
113. printf("Process %d terminated with signal %d\n", child\_pid, WTERMSIG(status));
114. **else** **if** (WIFSTOPPED(status))
115. printf("Process %d stopped due signal %d\n", child\_pid, WSTOPSIG(status));
116. }
117. }
119. **void** createSharedMemory()
120. {
121. shm\_id = shmget(IPC\_PRIVATE, **sizeof**(**int**), IPC\_CREAT | PERMS);
122. **if** (shm\_id == -1)
123. {
124. perror("Err: shmget");
125. exit(1);
126. }
128. shm = shmat(shm\_id, 0, 0);
129. **if** (shm == (**void** \*) -1)
130. {
131. perror("Err: shmat");
132. exit(1);
133. }
134. }
135. **void** start\_read()
136. {
137. **if** (semop(sem\_id, CANREAD, CANREAD\_SIZE) == -1)
138. {
139. perror("semop");
140. exit(1);
141. }
143. **if** (semop(sem\_id, STARTREAD, STARTREAD\_SIZE) == -1)
144. {
145. perror("semop");
146. exit(1);
147. }
148. }
150. **void** stop\_read()
151. {
152. **if** (semop(sem\_id, STOPREAD, STOPREAD\_SIZE) == -1)
153. {
154. perror("semop");
155. exit(1);
156. }
157. }
159. **void** start\_write()
160. {
161. **if** (semop(sem\_id, CANWRITE, CANWRITE\_SIZE) == -1)
162. {
163. perror("semop");
164. exit(1);
165. }
166. **if** (semop(sem\_id, STARTWRITE, STARTWRITE\_SIZE) == -1)
167. {
168. perror("semop");
169. exit(1);
170. }
171. }
173. **void** stop\_write()
174. {
175. **if** (semop(sem\_id, stopwrite, stopwrite\_SIZE) == -1)
176. {
177. perror("semop");
178. exit(1);
179. }
180. }
182. **void** reader(**int** id)
183. {
184. **while** (1)
185. {
186. sleep(randint(1, 3));
187. start\_read();
188. printf("\t\t\t\tReader-%d (pid %d) read %d\n", id, getpid(), \*shm);
189. stop\_read();
190. }
191. }
193. **void** writer(**int** id)
194. {
195. **while** (1)
196. {
197. sleep(randint(1, 2));
198. start\_write();
199. ++\*shm;
200. printf("Writer-%d (pid %d) wrote %d\n", id, getpid(), \*shm);
201. stop\_write();
202. }
203. }
205. **int** main()
206. {
207. initSemaphore();
208. createSharedMemory();
210. forkChildren(WRITER\_COUNT, writer);
211. forkChildren(READER\_COUNT, reader);
213. waitChildren(WRITER\_COUNT + READER\_COUNT);
215. shmctl(shm\_id, IPC\_RMID, NULL);
216. semctl(sem\_id, SEM\_ACTIVE\_WRITERS, IPC\_RMID, 0);
217. }

