

Отчет по курсу “Операционные системы”

Задание 4.02

Владимир Руцкий, 4057/2

24 декабря 2009 г.

Постановка задачи

“Напишите программу, моделирующую решение задачи читателей и писателей в условиях, когда разрешено одновременное чтение данных, а запись требует монопольного доступа.”

Выбранный метод решения

Для работы с потоками используются потоки POSIX.

Используется два мутекса:

- **databaseMutex** — взаимно исключающий доступ к базе данных. Им обладает либо один из писателей, который в данный момент пишет в базу, либо группа читателей, которые параллельно читают.
- **nReadersMutex** — взаимно исключающий доступ к счетчику читателей.

При запуске программа создаёт по одному потоку для каждого из читателей и писателей (**pthread_create**), дальше они работают независимо друг от друга, координируя действия только через мутексы.

Работа писателя:

1. Захват **databaseMutex** (**pthread_mutex_lock**).
2. Работа с базой данных.
3. Освобождение **databaseMutex** (**pthread_mutex_unlock**).

Работа читателя:

1. Захват **nReadersMutex**.
2. Если количество читателей равно нулю — текущий читатель единственный, а значит доступ к базе не захвачен — захват **databaseMutex**.
3. Увеличение числа читателей.
4. Освобождение **nReadersMutex**.

5. Чтение базы.
6. Захват **nReadersMutex**.
7. Если количество читателей равно одному — читатель был последним, доступ к базе данных больше не нужен — освобождение **databaseMutex**.
8. Уменьшение числа читателей.
9. Освобождение **nReadersMutex**.

Исходный код

Исходный код 1: task_4_02.c

```
1  /* task_4_02.c
2  * Task 4.02 on Unix course.
3  * Vladimir Rutsky <altsysrq@gmail.com>
4  * 24.12.2009
5  */
6
7  /* Solving Readers-writers problem using POSIX threads,
8  * mutexes and semaphores.
9  */
10
11 #include <stddef.h>
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <string.h>
15 #include <errno.h>
16
17 #include <unistd.h>
18 #include <pthread.h>
19
20 /*****
21  * Constants.
22  *****/
23
24 #define NREADERS 5
25 #define NWRITERS 2
26
27 static int const writerMaxPeriod = 10;
28 static int const writingMaxPeriod = 5;
29 static int const readerMaxPeriod = 30;
30 static int const readingMaxPeriod = 5;
31 static int const randomSleepsPeriod = 3;
32
33 /*****
34  * Used special types.
35  *****/
36
37 typedef void * (*thread_routine_type)( void * );
38 typedef struct
39 {
40     pthread_t threadId;
```

```

41  int userId;
42 } thread_info_type;
43
44 *****
45 * Common data.
46 *****/
47
48 /* Database mutual exclusive access */
49 static pthread_mutex_t databaseMutex = PTHREAD_MUTEX_INITIALIZER;
50
51 /* Number of currently reading database Readers */
52 static pthread_mutex_t nReadersMutex = PTHREAD_MUTEX_INITIALIZER;
53 static int nReaders = 0;
54
55 *****
56 * Reader and Writer implementations.
57 *****/
58
59 static void *writer( thread_info_type *info )
60 {
61     fprintf(stdout, "Writer_#####%d_started.\n", info->userId);
62
63     while (1)
64     {
65         sleep(rand() % writerMaxPeriod);
66         fprintf(stdout, "Writer_#####%d_attempting_to_write...\n", info->
            userId);
67
68         fprintf(stdout, "Writer_#####%d_waiting_for_DB_lock...\n", info->
            userId);
69         pthread_mutex_lock(&databaseMutex);
70         fprintf(stdout, "#####Writer_#####%d_got_DB_lock..Writing...\n", info->
            userId);
71
72         /* Actual writing in database... */
73         sleep(rand() % writingMaxPeriod);
74
75         fprintf(stdout, "<###Writer_#####%d_done_writing..Releasing_DB_lock.\n",
            info->userId);
76
77         pthread_mutex_unlock(&databaseMutex);
78     }
79
80     return NULL;
81 }
82
83 static void *reader( thread_info_type *info )
84 {
85     fprintf(stdout, "Reader_#####%d_started.\n", info->userId);
86
87     while (1)
88     {
89         sleep(rand() % writerMaxPeriod);
90         fprintf(stdout, "Reader_#####%d_attempting_to_read...\n", info->
            userId);
91

```

```

92     fprintf(stdout, "%d->Reader %d waiting for readers_count lock (before_read)...\n", info->userId);
93     pthread_mutex_lock(&nReadersMutex);
94     fprintf(stdout, "%d->Reader %d got readers_count lock.\n", info->
95         userId);
96     if (nReaders == 0)
97     {
98         fprintf(stdout, "%d->Reader %d is first accessing DB. Waiting for DB lock...\n", info->userId);
99         pthread_mutex_lock(&databaseMutex);
100         fprintf(stdout, "#####Reader %d locked DB.\n", info->userId);
101     }
102     ++nReaders;
103     fprintf(stdout, "%d->Reader %d unlocking readers_count.\n", info->
104         userId);
105     pthread_mutex_unlock(&nReadersMutex);
106
107     /* Actual reading of database... */
108     sleep(rand() % readingMaxPeriod);
109
110     fprintf(stdout, "%d->Reader %d waiting for readers_count lock (after read)...\n", info->userId);
111     pthread_mutex_lock(&nReadersMutex);
112     fprintf(stdout, "%d->Reader %d got readers_count lock.\n", info->
113         userId);
114     --nReaders;
115     if (nReaders == 0)
116     {
117         fprintf(stdout, "%d->Reader %d was last accessing DB. Releasing DB lock.\n", info->userId);
118         pthread_mutex_unlock(&databaseMutex);
119     }
120     fprintf(stdout, "%d->Reader %d unlocking readers_count.\n", info->
121         userId);
122     pthread_mutex_unlock(&nReadersMutex);
123 }
124
125 return NULL;
126 }
127
128 /* The main program function */
129 int main( int argc, char const *argv[] )
130 {
131     int i;
132     thread_info_type threads[NREADERS + NWRITERS];
133     int nextThreadId = 0;
134
135     /* Initializing locks to common resources */
136     srand(0);
137
138     /* Creating writers */
139     for (i = 0; i < NWRITERS; ++i)
140     {
141         threads[nextThreadId].userId = i + 1;
142         if (pthread_create(&threads[nextThreadId].threadId, NULL, (
143             thread_routine_type)writer, (void *)&threads[nextThreadId]) != 0)

```

```

139     {
140         perror("pthread_create");
141         fprintf(stderr, "Error: pthread_create() failed.\n");
142         return 1;
143     }
144
145     ++nextThreadId;
146 }
147
148 /* Creating readers */
149 for (i = 0; i < NREADERS; ++i)
150 {
151
152     threads[nextThreadId].userId = i + 1;
153     if (pthread_create(&threads[nextThreadId].threadId, NULL, (
154         thread_routine_type)reader, (void *)&threads[nextThreadId]) != 0)
155     {
156         perror("pthread_create");
157         fprintf(stderr, "Error: pthread_create() failed.\n");
158         return 1;
159     }
160
161     ++nextThreadId;
162 }
163
164 /* Get threads time to work */
165 while (1)
166     ;
167
168 return 0;
169 }

```