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

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