Создание многопоточных приложений на основе POSIX threads

Контрольная работа по курсу
«Системы реального времени»

Выполнил: ст. гр. 4032 Турбин А.
Проверил: доц. Благодаров А. В.

РГРТУ, 2007

Введение

В данной работе для реализации многопоточности используется программный интерфейс (API) POSIX threads. Этот интерфейс является стандартным (международный стандарт IEEE Std 1003.1c-1995) и широко используется в UNIX-подобных операционных системах (в частности, в ОС Linux). Существуют также реализации для семейства ОС Windows. Интерфейс POSIX threads насчитывает несколько десятков функций, которые позволяют контролировать все аспекты выполнения многопоточных приложений. Но для выполнения задания в данной работе в основном достаточно двух функции: `pthread_create` и `pthread_join`.

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
        void *(*start_routine)(void*), void *arg);

int pthread_join(pthread_t thread, void **value_ptr);
```

Функция `pthread_create` создает асинхронный поток (тред) и сразу же возвращает управление в основной процесс. Асинхронный поток будет выполняться независимо. Для выполнения асинхронного потока создается отдельный стековый сегмент памяти и управляющая структура ОС (регистры, счётчик команд и т. д.), но сегмент общей памяти (в частности, глобальные переменные программы) являются одинаково доступными для всех тредов. Функция `pthread_create` возвращает идентификатор вновь созданного треда через указатель `pthread_t *thread`. Аргумент `pthread_attr_t *attr` позволяет указать дополнительные параметры (характеристики) треда; для создания треда с параметрами по умолчанию можно передать в `attr` значение `NULL`. Указатель на функцию `start_routine` определяет *точку входа* для исполнения в асинхронном потоке, то есть в треде будет выполняться код функции, которая специфицирована через указатель на функцию `start_routine`. Для этой функции в качестве формального параметра предусмотрен нетипизированный указатель, который передаётся через `void *arg`. Функция также может через нетипизированный указатель возвращать некоторое значение. Исполнение вновь созданного треда заканчивается, когда функция, специфицированная через аргумент `start_routine`, возвращает управление.

Функция `pthread_join` дожидается завершения выполнения треда. В качестве первого аргумента этой функции нужно передать идентификатор ранее созданного треда. Второй аргумент позволяет получить доступ к значению, возвращаемому функцией `start_routine` (в данной работе доступ к значению реализован подругому, а в качестве аргумента `void **value_ptr` используется `NULL`).

Ненулевое возвращаемое значение функций `pthread_create` и `pthread_join` сигнализирует об ошибке.

Таким образом, функции `pthread_create` и `pthread_join` позволяют реализовать следующий примитив параллельных вычислений и синхронизации:

```
/* распараллеливание: запуск асинхронных потоков */
pthread_create(&thread1, NULL, func1, arg1);
pthread_create(&thread2, NULL, func2, arg2);

/* синхронизация: ожидание завершения потоков */
pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
```

1. Задание

Вариант задания №12.

Создать многопоточное приложение, максимально распараллеливающее вычисление выражения

$$T = U + V + W,$$

где

$$U = e^{-AC}, \quad V = \sqrt{C^2 + A^2}, \quad W = A + C, \quad C = \cos^7 A,$$

$$A = \sum_{i=1}^{\infty} \frac{(-1)^i \cdot x^{2i}}{(2i)!} \quad \text{с погрешностью } \varepsilon = 10^{-6}, \text{ где } |x| < 1,$$

$$B = \prod_{k=1}^{50} \left(1 + \frac{y^2}{2k}\right), \quad \text{где } |y| < 1.$$

2. Анализ задания и проектирование программы

Распараллеливание вычислений

В соответствие с заданием и используемыми примитивами распараллеливание вычислений можно описать следующим образом.

Вычисление A, B и C:

1. Запустить вычисление суммы A.
2. Запустить вычисление произведения B.
3. Дождаться завершения вычисления суммы A.
4. Запустить вычисление выражения C.
5. Дождаться завершения вычисления произведения B.
6. Дождаться завершения вычисления выражения C.

Вычисление U, V, и W: эти значения могут быть вычислены полностью независимо друг от друга, поэтому дополнительной синхронизации не требуется. Вместе с тем, в каждом из этих выражений используется одно из значений A, B или C, поэтому предыдущий этап вычислений должен быть полностью завершён.

Таким образом, для максимального распараллеливания вычислений в программе нужно создать шесть потоков, для вычисления соответственно A, B, C; U, V, и W. При этом наибольшее распараллеливание вычислений достигается на втором этапе, когда одновременно могут выполняться потоки вычислений U, V, и W.

Механизм передачи аргументов и возврата значения

Основные вычисления программы удобно представить в виде «математических функций». Например, вычисление суммы A можно представить в виде функции от параметра x:

```
/* вычисление суммы A */
double sumA(double x);
```

С другой стороны, для организации многопоточных вычислений функции такого вида нельзя использовать в чистом виде, так как для выполнения в асинхронном потоке можно передать только функции вида

```
void *thread_sumA(void *arg);
```

Мы реализуем несложный механизм преобразования формальных аргументов «математических функций» к бестиповому указателю, используя промежуточную структуру данных и функцию-«обёртку». Для математических функций с одним аргументом используется структура данных

```
struct thread_arg_ret
{
        double arg; /* аргумент функции */
        double *ret; /* по какому адресу сохранить результат */
};
```

Функция-«обёртка» позволяет «полуавтоматически» (при помощи макроса) сформировать из «математической» функции `sumA` «тредовую» функцию `thread_sumA`, которая в качестве аргумента получает указатель на структуру `thread_arg_ret` и записывает вычисленное значение по адресу `ret`. Теперь функцию `thread_sumA` можно использовать непосредственно в качестве «точки входа» в поток, а её «аргумент» —указатель `thread_arg_ret` — можно безопасно передавать через бестиповый указатель.

Аналогично реализована дополнительная структура данных и функция-«обёртка» для математических функций двух аргументов.

Вычисление суммы A

Требуется вычислить

$$A = \sum_{i=1}^{\infty} \frac{(-1)^i \cdot x^{2i}}{(2i)!},$$

с погрешностью $\varepsilon = 10^{-6}$, где $|x| < 1$.

Вычислять каждый очередной член суммы будет рекуррентно, то есть через предыдущее значение. Определим переходный коэффициент от i-го члена суммы к $i+1$ члену суммы:

$$\frac{(-1)^{(i+1)} \cdot x^{2(i+1)}}{(2(i+1))!} \Big/ \frac{(-1)^i \cdot x^{2i}}{(2i)!} = \frac{-x^2}{(2i+1)(2i+2)}.$$

Вычислим первый член суммы:

$$\left.\frac{(-1)^i \cdot x^{2i}}{(2i)!}\right|_{i=1} = -\frac{x^2}{2}.$$

3. Текст программы

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <assert.h>
```

```
5  #include <pthread.h>
6
7  /* погрешность вычислений */
8  #define eps 10e-6
9
10 /* вычисление суммы A */
11 double sumA(double x)
12 {
13         /* по условию |x| < 1, иначе сумма расходится */
14         assert(abs(x) < 1);
15         /* параметр цикла */
16         int i = 1;
17         /* очередной член суммы */
18         /* начальное значение - первый член */
19         double v = -x*x/2;
20         /* вычисляемая сумма */
21         /* в начале она равна первому члену */
22         double sum = v;
23         while (1) {
24                 /* итерация начинается со второго члена */
25                 ++i;
26                 /* домножаем член на переходный коэффициент */
27                 v *= -x*x/(2*i+1)/(2*i+2);
28                 /* добавляем новый член к сумме */
29                 sum += v;
30                 /* если новый член меньше eps, то будем считать,
31                  * что заданная точность достигнута */
32                 if (v < eps)
33                         break;
34         }
35         return sum;
36 }
37
38 /* вычисление произведения B */
39 double prodB(double y)
40 {
41         assert(abs(y) < 1);
42         int k;
43         double prod = 1;
44         for (k=1; k <= 50; ++k)
45                 prod *= 1 + y*y/2/k;
46         return prod;
47 }
48
49 /* вычисление выражения C */
50 double calcC(double A)
51 {
52         /* cos^7(A) */
53         double c = cos(A);
54         double c3 = c*c*c;
55         return c*c3*c3;
56 }
```

```
57
58 /* вычисление выражений U, V и W */
59 double calcU(double A, double C)
60 {
61         return exp(-A*C);
62 }
63 double calcV(double C, double A)
64 {
65         return sqrt(C*C+A*A);
66 }
67 double calcW(double A, double C)
68 {
69         return sqrt(A+C);
70 }
71
72 /* дополнительная структура данных для работы
73  * с математическими функциями в тредах */
74 struct thread_arg_ret
75 {
76         double arg; /* аргумент функции */
77         double *ret; /* по какому адресу сохранить результат */
78 };
79
80 /* Функция-обёртка для вызова математических функций из тредов */
81 void *thread_func(double (*f)(double), struct thread_arg_ret *x)
82 {
83         *(x->ret) = f(x->arg);
84         return (void*)0;
85 }
86
87 /* макрос для создания обёрток на основе thread_func */
88 #define make_thread_func(thr,f) \
89         void *thr(struct thread_arg_ret *x) { \
90                 return thread_func(f, x); \
91         }
92
93 /* теперь для каждой математической функции можно задать тред */
94 make_thread_func(thread_func_A, sumA);
95 make_thread_func(thread_func_B, prodB);
96 make_thread_func(thread_func_C, calcC);
97
98 /* аналогичные обёртки для функций двух аргументов */
99 struct thread_arg2_ret
100 {
101         double arg1, arg2;
102         double *ret;
103 };
104 void *thread_func2(double (*f)(double, double), struct thread_arg2_ret *x)
105 {
106         *(x->ret) = f(x->arg1, x->arg2);
107         return (void*)0;
108 }
```

```
109 #define make_thread_func2(thr,f) \
110         void *thr(struct thread_arg2_ret *x) { \
111                 return thread_func2(f, x); \
112         }
113 make_thread_func2(thread_func_U, calcU);
114 make_thread_func2(thread_func_V, calcV);
115 make_thread_func2(thread_func_W, calcW);
116
117 int main(int argc, const char *argv[])
118 {
119         /* основные параметры задачи */
120         double x, y;
121         /* инициализируем основные параметры задачи
122          * через позиционные аргументы командной строки */
123         assert(argc - 1 == 2);
124         x = atof(argv[1]);
125         y = atof(argv[2]);
126         /* отладочная печать x и y */
127         printf("x=%f\n", x);
128         printf("y=%f\n", y);
129
130         /* основные переменные задачи */
131         double A, B, C, U, V, W;
132
133         /* основной результат задачи */
134         double T;
135
136         /* переменная для проверки ошибок */
137         int rc;
138
139         /* ВЫЧИСЛЕНИЕ СУММЫ A */
140         /* идентификатор треда */
141         pthread_t thread_A;
142         /* x - это аргумент функции, результат записать в A */
143         struct thread_arg_ret arg_ret_A = {x, &A};
144         /* запуск треда */
145         rc = pthread_create(&thread_A, NULL, thread_func_A, &arg_ret_A);
146         /* убедимся, что тред запустился */
147         assert(rc == 0);
148
149         /* ВЫЧИСЛЕНИЕ ПРОИЗВЕДЕНИЯ B (аналогично) */
150         pthread_t thread_B;
151         struct thread_arg_ret arg_ret_B = {y, &B};
152         rc = pthread_create(&thread_B, NULL, thread_func_B, &arg_ret_B);
153         assert(rc == 0);
154
155         /* теперь выполняются thread_A и thread_B */
156         /* нужно дождаться завершения thread_A */
157         pthread_join(thread_A, NULL);
158
159         /* теперь можно запустить ВЫЧИСЛЕНИЕ C */
160         pthread_t thread_C;
```

```
161         struct thread_arg_ret arg_ret_C = {A, &C};
162         rc = pthread_create(&thread_C, NULL, thread_func_C, &arg_ret_C);
163
164         /* дождаться завершения thread_B и thread_C */
165         pthread_join(thread_B, NULL);
166         pthread_join(thread_C, NULL);
167
168         /* ПЕРВЫЙ ЭТАП ВЫЧИСЛЕНИЙ ЗАКОНЧЕН */
169         /* значения A, B и C полностью вычислены */
170         printf("A=%f\n", A);
171         printf("B=%f\n", B);
172         printf("C=%f\n", C);
173
174         /* Второй этап вычислений - U, V и W */
175         pthread_t thread_U, thread_V, thread_W;
176         struct thread_arg2_ret
177                 arg2_ret_U = { A, C, &U },
178                 arg2_ret_V = { C, A, &V },
179                 arg2_ret_W = { A, C, &W };
180         rc = pthread_create(&thread_U, NULL, thread_func_U, &arg2_ret_U);
181         rc = pthread_create(&thread_V, NULL, thread_func_V, &arg2_ret_V);
182         rc = pthread_create(&thread_W, NULL, thread_func_W, &arg2_ret_W);
183         /* ждать окончания вычислений U, V и W */
184         pthread_join(thread_U, NULL);
185         pthread_join(thread_V, NULL);
186         pthread_join(thread_W, NULL);
187         printf("U=%f\n", U);
188         printf("V=%f\n", V);
189         printf("W=%f\n", W);
190
191         /* окончательный результат */
192         T = U + V + W;
193         printf("T=%f\n", T);
194         return 0;
195 }
```

4. Проверка работы программы

Напомним, что параметры задачи x и y передаются через позиционные аргументы командной строки (соответственно первый и второй). Далее приведён протокол работы программы с параметрами $x = 0.3$ и $y = 0.5$.

```
OS> srv 0.3 0.5
x=0.300000
y=0.500000
A=-0.044865
B=1.733992
C=0.992977
U=1.045557
V=0.993990
W=0.973710
T=3.013258
```