Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет: “Информационные технологии и прикладная математика”
Кафедра: 806 “Вычислительная математика и программирование”

Лабораторная работа №3
по курсу “Операционные системы”

Студент	Полей-Добронравова А.В.
Группа	М8О-207Б-18
Вариант	7
Преподаватель	Миронов Е.С.
Дата	15.03.2020
Оценка	

Москва, 2020

1. Постановка задачи

Цель работы
Целью является приобретение практических навыков в:

• Управление потоками в ОС

• Обеспечение синхронизации между потоками

Задание

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы (Windows/Unix). При создании необходимо предусмотреть ключи, которые позволяли бы задать максимальное количество потоков, используемое программой. При возможности необходимо использовать максимальное количество возможных потоков. Ограничение потоков может быть задано или ключом запуска вашей программы, или алгоритмом. Так же необходимо уметь продемонстрировать количество потоков, используемое вашей программой с помощью стандартных средств операционной системы.

7. Произвести распараллеленный поиск по дереву общего вида. Первоначальные данные задаются в файле в виде простых команд (например, "+ 1 /" добавить элемент к корню дерева, "+ 2 1/3/" добавить элемент 2 по пути 1->3 в дереве).(ограничения по потокам нет)

2. Решение задачи

Единицу выполнения кода в виде объекта — это и есть поток.

Разница между потоками и процессами

- Потоки используют память, выделенную под процесс, а процессы требуют себе отдельное место в памяти. Поэтому потоки создаются

и завершаются быстрее: системе не нужно каждый раз выделять им новое адресное пространство, а потом высвобождать его.

- Процессы работают каждый со своими данными — обмениваться чем-то они могут только через механизм межпроцессного взаимодействия. Потоки обращаются к данным и ресурсам друг друга напрямую: что изменил один — сразу доступно всем. Поток может контролировать «собратьев» по процессу, в то время как процесс контролирует исключительно своих «дочек». Поэтому переключаться между потоками быстрее и коммуникация между ними организована проще.
- В Linux у каждого процесса есть идентификатор. Есть он, естественно, и у процессов-потоков. С другой стороны, спецификация POSIX 1003.1c требует, чтобы все потоки многопоточного приложения имели один идентификатор. Вызвано это требование тем, что для многих функций системы многопоточное приложение должно представляться как один процесс с одним идентификатором. Проблема единого идентификатора решается в Linux весьма элегантно. Процессы многопоточного приложения группируются в группы потоков (thread groups). Группе присваивается идентификатор, соответствующий идентификатору первого процесса многопоточного приложения. Именно этот идентификатор группы потоков используется при «общении» с многопоточным приложением. Функция getpid(2), возвращает значение идентификатора группы потока, независимо от того, из какого потока она вызвана. Функции kill() waitpid() и им подобные по

умолчанию также используют идентификаторы групп потоков, а не отдельных процессов. Получить идентификатор потока (thread ID) можно с помощью функции gettid(2), однако саму функцию нужно еще определить с помощью макроса _syscall.

Использованные системные вызовы:

int pthread_create(*pthread_t, const pthread_attr_t *attr, void* (*start_routine)(void*), void *arg); - Создание потока.

Функция получает в качестве аргументов указатель на поток, переменную типа pthread_t, в которую, в случае удачного завершения сохраняет id потока. pthread_attr_t – атрибуты потока. В случае если используются атрибуты по умолчанию, то можно передавать NULL. start_routin – это непосредственно та функция, которая будет выполняться в новом потоке. arg – это аргументы, которые будут переданы функции. В случае успешного выполнения функция возвращает 0. Если произошли ошибки, то могут быть возвращены следующие значения

EAGAIN – у системы нет ресурсов для создания нового потока, или система не может больше создавать потоков, так как количество потоков превысило значение PTHREAD_THREADS_MAX

EINVAL – неправильные атрибуты потока (переданные аргументом attr)

EPERM – Вызывающий поток не имеет должных прав для того, чтобы задать нужные параметры или политики планировщика.

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

Откладывает выполнение вызывающего (эту функцию) потока, до тех пор, пока не будет выполнен поток thread. Когда pthread_join выполнилась успешно, то она возвращает 0. Если поток явно вернул значение, то оно будет помещено в переменную value_ptr. Возможные ошибки, которые возвращает pthread_join

EINVAL – thread указывает на не объединяемый поток

ESRCH – не существует потока с таким идентификатором, который хранит переменная thread

EDEADLK – был обнаружен дедлок (взаимная блокировка), или же в качестве объединяемого потока указан сам вызывающий поток.

ssize_t read(int fd, void* buf, size_t count) - считывает из файла, на который ссылается файловый дескриптор fd, count байт в буфер, начинающийся по адресу buf.

int close(int fd) - закрывает файловый дескриптор.

int open(const char* s) - открывает файл.

Так же из библиотеки pthread.h использованы

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
pthread_mutex_lock(&mutex);
pthread_mutex_unlock(&mutex);
pthread_mutex_destroy(&mutex);
```

В моей программе в начале формируется дерево в функции

void create_tree(**struct** tree* t,**const char*** s)

считыванием из файла с использованием

struct tree* find1 (**struct** tree* t, **int** a) *// есть ли шаг из текущего узла в узел со значением a*

С выводом ошибок о неправильно заданном пути. После этого вводится значение узла, для которого должен выполниться поиск по дереву. Сам поиск реализован в функции, которую вызывают же и потоки при распараллеливании поиска **void*** search_tree(**void*** k). В нее передаётся указатель на структуру данных

```
struct i_t_s {
    struct tree* t;
    int a;
    int path[16];
    int pos;
};
```

Где t - указатель на текущий узел, a - искомое значение узла, path[16] массив, содержащий значения вершин на пути к текущему узлу, pos - первый свободный индекс в path[16].

Если значение текущей вершины совпадает с искомой, мьютексом заблокировать редактирование общих данных процесса, присвоить общей переменной answer указатель на текущую вершину, вывести путь до нее. Если нет, создать потоки для поиска в левом и правом поддереве. Ожидать завершения этих потоков, по завершению вернуть значение answer. Т.к. ожидается завершение всех потоков программы, найдутся все находящиеся в дереве значения, совпадающие с искомым, answer будет иметь указатель на одну из них, но пользователь получит пути до каждого из них.

Демонстрация работы программы:

Тест 1

```
MacBook-Pro-Amelia:Documents amelia$ gcc -o l3 os3.c
MacBook-Pro-Amelia:Documents amelia$ sudo dtruss ./l3 os3.txt

Starting creating tree
Read line
+ 1 / ??`?
Inserted in top

 Tree now
1

Read line
+ 2 1/3/
Error wrong path on 3
```

Tree now
1

Read line
+ 2 1/
Inserted

Tree now
1
 2

Read line
+ 3 1/2/
Inserted

Tree now
1
 2
  3

Read line
+ 4 1/2/
Inserted

Tree now
1
 2
  3
  4

Read line
+ 4 1/
Inserted

Tree now
1
 2
  3
  4
 4

Read line

```
+ 5 1/2/4/
Inserted

 Tree now
1
 2
  3
  4
   5
 4

1
 2
  3
  4
   5
 4

 Enter number for searching
4
4 in path: 1 /
4 in path: 1 / 2 /

 There is it in the tree
```

Тест 2

```
MacBook-Pro-Amelia:Documents amelia$ ./l3 os3.txt

Starting creating tree
Read line
+ 1 / ??`?
Inserted in top

 Tree now
1

Read line
+ 2 1/3/
Error wrong path on 3

 Tree now
```

```
1

Read line
+ 2 1/
Inserted

 Tree now
1
 2

Read line
+ 3 1/2/
Inserted

 Tree now
1
 2
  3

Read line
+ 4 1/2/
Inserted

 Tree now
1
 2
  3
  4

Read line
+ 4 1/
Inserted

 Tree now
1
 2
  3
  4
 4

Read line
+ 5 1/2/4/
```

Inserted

Tree now
1
2
3
4
5
4

1
2
3
4
5
4

Enter number for searching
6

There is NOT it in the tree

3. Листинг программы

```
#include <stdio.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <signal.h>
#include <pthread.h>

struct tree {
   int node;
   struct tree* l;
   struct tree* r;
};
struct i_t_s {
   struct tree* t;
   int a;
   int path[16];
   int pos;
```

```
};

struct tree* answer = NULL;

struct tree* find1 (struct tree* t, int a) { // есть ли шаг из текущего узла в узел со значением a
   if (t == NULL) {
      return NULL;
   }
   else {
      if (t->node == a) {
         return t;
      }
      else {
         while (t->r != NULL) {
            if (t->r->node == a) {
               return t->r;
            }
            else {
               t = t->r;
            }
         }
      }
      return NULL;
   }
}

void* search_tree(void* k) {
   struct tree* b;
   struct i_t_s* p = (struct i_t_s*)k;
   b = p->t;
   pthread_mutex_t mutex;
   pthread_mutex_init(&mutex, NULL);
   if (b != NULL) { //Пока не встретится пустой узел
      if (b->node == p->a) {
         pthread_mutex_lock(&mutex);
         answer = p->t;
         pthread_mutex_unlock(&mutex);
         printf("%d in path:", p->a);
         for (int y = 0; y <= p->pos; y++) {
            printf(" %d /", p->path[y]);
         }
         printf("\n");
         return answer;
      }
      else {
         pthread_t thread1, thread2;
         pthread_attr_t attr;
         pthread_attr_init(&attr);
         int result1, result2;
         struct i_t_s* j1 = malloc(sizeof(struct i_t_s));
         j1->t = p->t->l;
         j1->a = p->a;
         for (int y = 0; y <= p->pos; y++) {
            j1->path[y] = p->path[y];
```

```
            }
            j1->pos = p->pos + 1;
            j1->path[j1->pos] = p->t->node;
            struct i_t_s* j2 = malloc(sizeof(struct i_t_s));
            j2->t = p->t->r;
            j2->a = p->a;
            for (int y = 0; y <= p->pos; y++) {
               j2->path[y] = p->path[y];
            }
            j2->pos = p->pos;
            for (int y = 0; y <= p->pos; y++) {
               j2->path[y] = p->path[y];
            }
            result1 = pthread_create(&thread1, NULL, search_tree, (void*)j1); //функция для левого поддерева
            result2 = pthread_create(&thread2, NULL, search_tree, (void*)j2); //функция для правого поддерева
            if (result1 != 0) {
               perror("Joining the thread");
               pthread_mutex_destroy(&mutex);
               return NULL;
            }
            if (result2 != 0) {
               perror("Joining the thread");
               pthread_mutex_destroy(&mutex);
               return NULL;
            }
            int status_addr1, status_addr2;
            int status1 = pthread_join(thread1, (void**)&status_addr1);
            if (status1 != 0) {
               printf("main error: can't join thread, status = %d\n", status1);
               exit(-1);
            }
            int status2 = pthread_join(thread2, (void**)&status_addr2);
            if (status2 != 0) {
               printf("main error: can't join thread, status = %d\n", status2);
               exit(-1);
            }
         }
      }
      pthread_mutex_destroy(&mutex);
      return answer;
   }
   void print_tree (struct tree* t, int i) {
      if (t != NULL) {
         for (int j = 0; j < i; j++) {
            printf(" ");
         }
         printf("%d\n", t->node);
         print_tree(t->l, i+1);
         print_tree(t->r, i);
      }
   }
   void create_tree(struct tree* t,const char* s) {
      int fd, i,j = 0, vst, p1, fl, fl2;
      struct tree* b, *where;
```

```
char buf[16], c;
printf("Starting creating tree \n");
fd = open(s, O_RDONLY);
while(read(fd, &c, sizeof(char)) > 0) {
   if (c != '\n' && j < 16) {
      buf[j] = c;
      j++;
   }
   else {
      j = 0;
   printf("Read line \n");
      for (i = 0; i < 16; i++) {
         printf("%c", buf[i]);
      }
      printf("\n");
   i = 0;
   while (i < 16) {
      while (buf[i] == ' ' && i < 16) {
         i++;
      }
   if (buf[i] == '+') {
      fl = 1;
      i++;
      while (buf[i] == ' ' && i < 16) {
         i++;
      }
      vst = 0;
     while (buf[i] != ' ' && i < 16) {
         vst = vst * 10 + buf[i] - '0';
         i++;
      }
      while (buf[i] == ' ' && i < 16) {
         i++;
      }
      struct tree* a = malloc(sizeof(struct tree));
      a->l = t;
      where = a;
      fl2 = 0;
      p1 = 0;
      while(buf[i] != ' ' || i >= 16) {
      if (buf[i] == '/' && buf[i + 1] == ' ') {
         break;
      }
      else {
         p1 = 0;
         fl2 = 2;
         while (buf[i] != '/' && i < 16) {
            p1 = p1 * 10 + buf[i] - '0';
            i++;
         }
         where = where->l;
         where = find1(where, p1);
         if (where == NULL) {
            printf("Error wrong path on %d\n", p1);
```

```
                while (buf[i] != ' ' && i < 16) {
                    i++;
                }
                fl2 = -1;
                break;
            }
            i++;
        }
        }
            i++;
            b = malloc(sizeof(struct tree));
            b->node = vst;
            b->l = NULL;
            b->r = NULL;
            if (fl2 > 0) {
                if (where->l != NULL) {
                    where = where->l;
                while (where->r != NULL) {
                    where = where->r;
                }
                    where->r = b;
                }
                else {
                    where->l = b;
                }
                i++;
            printf("Inserted \n");
            } else if (fl2 == 0) {
                t->node = b->node;
                t->l = NULL;
                t->r = NULL;
                printf("Inserted in top\n");
            }
                printf("\n Tree now \n");
                print_tree(t, 0);
                printf("\n");
            for (i = 0; i < 16; i++) {
                buf[i] = ' ';
            }
      }
      }
      }
   }
}

int main(int argc, const char * argv[]) {
    struct tree* t = malloc(sizeof(struct tree));
    t->l = NULL;
    t->r = NULL;
    void* answer;
    int a;
    create_tree(t, argv[1]);
    print_tree(t, 0);
    printf("\n Enter number for searching \n");
```

```
    scanf("%d", &a);
    struct i_t_s* j1 = malloc(sizeof(struct i_t_s));
    j1->t = t;
    j1->a = a;
    j1->path[0] = t->node;
    j1->pos = -1;
    answer = search_tree((void*)j1);
    if (answer != NULL) {
        printf("\n There is it in the tree \n");
    }
    else {
        printf("\n There is NOT it in the tree \n");
    }
    return 0;
}
```

4. Вывод

Современные операционные системы и микропроцессоры уже давно поддерживает многозадачность и вместе с тем, каждая из этих задач может выполняться в несколько потоков. Это дает ощутимый прирост производительности вычислений и позволяет лучше масштабировать пользовательские приложения и сервера, но за это приходится платить цену — усложняется разработка программы и ее отладка.

Прежде чем приступать к программированию потоков, следует ответить на вопрос, а нужны ли они вам. Мы уже знаем, насколько хорошо развиты в Linux средства межпроцессного взаимодействия. С помощью управления процессами в Linux можно решить многие задачи, которые в других ОС решаются только с помощью потоков. Потоки часто становятся источниками программных ошибок особого рода. Эти ошибки возникают при использовании потоками разделяемых ресурсов системы (например, общего адресного пространства) и являются частным случаем более широкого класса ошибок – ошибок синхронизации. Если задача разделена между независимыми процессами, то доступом к их общим ресурсам управляет операционная система, и вероятность ошибок из-за конфликтов доступа снижается. Впрочем, разделение задачи между несколькими независимыми процессами само по себе не защитит вас от других разновидностей ошибок синхронизации. В пользу потоков можно указать то, что накладные расходы на создание нового потока в многопоточном приложении ниже, чем накладные расходы на создание нового

самостоятельного процесса. Уровень контроля над потоками в многопоточном приложении выше, чем уровень контроля приложения над дочерними процессами. Кроме того, многопоточные программы не склонны оставлять за собой вереницы зомби или «осиротевших» независимых процессов.