Министерство науки и образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
"Московский институт электронной техники"
(МИЭТ)

Отчет по лабораторной работе № 5

Операционные системы

Выполнил: студент ПМ - 31

Мартынова Мария Олеговна

2023 г.

Задание 1

Скомпилировать mutex.c без использования и с использованием мьютекса. Объяснить разницу в поведении программы.

GCC - это свободно доступный оптимизирующий компилятор для языков C, C++.

Чтобы откомпилировать исходный код C++, находящийся в файле F.c, и создать объектный файл F.o, выполните команду:

gcc -c <compile-options> F.c

Здесь строка compile-options указывает возможные дополнительные опции компиляции.

-c – только компиляция. Из исходных файлов программы создаются объектные файлы в виде name.o. Компоновка не производится.

-Wall – вывод сообщений о всех предупреждениях или ошибках, возникающих во время трансляции программы.

-o – данная опция задает имя исполняемого файла.

Состояние гонки (англ. race condition), также конкуренция — ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода.

Под гонкой условий (*Condition Race*), которую также называют гонкой данных (*Data Race*) понимают ситуацию, когда два или более потока соперничают за обладание некоторым общим ресурсом. Чаще всего соперничество возникает из-за такого ресурса как оперативная *память*. Но таковым ресурсом может быть и внешняя *память* (работа с одним и тем же файлом, например), или некоторое устройство, подключенное к компьютеру.

Data race возникает при условии:

- два или более потока обращаются к одной и той же общей переменной;

- как минимум один из потоков пытается менять значение этой переменной;
- потоки не используют блокировки для обращения к этой переменной.

Критическая секция — участок исполняемого кода программы, в котором производится доступ к общему ресурсу (данным или устройству), который не должен быть одновременно использован более чем одним потоком выполнения. При нахождении в критической секции двух (или более) потоков возникает состояние «гонки» («состязания»). Во избежание данной ситуации необходимо выполнение четырех условий:

1. Два потока не должны одновременно находиться в критических областях.
2. В программе не должно быть предположений о скорости или количестве процессоров.
3. Поток, находящийся вне критической области, не может блокировать другие потоки.
4. Невозможна ситуация, в которой поток вечно ждет попадания в критическую область.

Критическая секция – это часть программы, исполнение которой может привести к возникновению *race condition* для определенного набора программ. Чтобы исключить эффект гонок *по* отношению к некоторому ресурсу, необходимо организовать работу так, чтобы в каждый момент времени только один процесс мог находиться в своей *критической секции*, связанной с этим ресурсом. Иными словами, необходимо обеспечить реализацию *взаимоисключения* для *критических секций* программ. Реализация *взаимоисключения* для *критических секций* программ с практической точки зрения означает, что *по* отношению к другим процессам, участвующим во взаимодействии, *критическая секция* начинает выполняться как *атомарная операция*.

В начале создается потоковая функция. Затем новый процесс создается функцией **pthread_create(),** объявленной в заголовочном файле pthread.h. Далее, вызывающая сторона продолжает выполнять какие-то свои действия параллельно потоковой функции.

При удачном завершении pthread_create() возвращает код 0, ненулевое значение сигнализирует об ошибке.

- Создание потока:

```
int pthread_create( pthread_t *thread, pthread_attr_t *attr,      void
  *(*thread_function)(void *), void *arg );
```

  - thread –указатель на идентификатор созданного потока
  - attr – атрибуты потока
  - third argument – функция, которую поток будет исполнять
  - arg – аргументы функции (обычно структура)
  - returns 0 for success

Функция **pthread_join()** ожидает завершения потока обозначенного THREAD_ID. Если этот поток к тому времени был уже завершен, то функция немедленно возвращает значение. Смысл функции в том, чтобы синхронизировать потоки.

- Ожидание завершения потока:

```
int pthread_join( pthread_t thread, void **thread_return )
```

  - Основной поток дожидается завершения потока с идентификатором *thread*
  - Второй аргумент – значение возвращаемое потоком
  - returns 0 for success
  - Следует всегда дожидаться завершения потока

Правильный способ при использовании pthreads - это скомпилировать и связать с помощью -pthread , который, помимо прочего, будет связан с библиотекой pthread.

```
doing another thing
counter = 38
doing another thing
counter = 39
doing one thing
counter = 39
doing another thing
counter = 40
doing another thing
counter = 41
doing another thing
counter = 42
doing one thing
counter = 40
doing one thing
counter = 41
doing one thing
counter = 42
doing one thing
counter = 43
doing another thing
counter = 43
doing another thing
counter = 44
doing one thing
counter = 44
doing another thing
counter = 45
doing one thing
counter = 45
doing one thing
counter = 46
doing one thing
counter = 47
doing one thing
counter = 48
All done, counter = 49
~/.../lab5/src$
```

```
doing one thing
counter = 82
doing another thing
counter = 83
doing another thing
counter = 84
doing another thing
counter = 85
doing another thing
counter = 86
doing one thing
counter = 87
doing one thing
counter = 88
doing one thing
counter = 89
doing one thing
counter = 90
doing another thing
counter = 91
doing another thing
counter = 92
doing another thing
counter = 93
doing another thing
counter = 94
doing another thing
counter = 95
doing another thing
counter = 96
doing one thing
counter = 97
doing another thing
counter = 98
doing another thing
counter = 99
All done, counter = 100
```

```c
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void do_one_thing(int *);
void do_another_thing(int *);
void do_wrap_up(int);
int common = 0; /* A shared variable for two threads */
int r1 = 0, r2 = 0, r3 = 0;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;

int main() {
  pthread_t thread1, thread2;

  if (pthread_create(&thread1, NULL, (void *)do_one_thing,
        (void *)&common) != 0) {//создание треда(первая переменная - записывается айдишник; некоторые специфические общии[обычно NULL]; третий - функция, которая будет вызвана для треда; параметр вызванной функции)
    perror("pthread_create");
    exit(1);
  }

  if (pthread_create(&thread2, NULL, (void *)do_another_thing,
                     (void *)&common) != 0) {
    perror("pthread_create");
    exit(1);
  }

  if (pthread_join(thread1, NULL) != 0) {// дожидается окончания треда(айдишник, значение, которое возвращает тред)
    perror("pthread_join");
    exit(1);
  }

  if (pthread_join(thread2, NULL) != 0) {
    perror("pthread_join");
    exit(1);
  }

  do_wrap_up(common);

  return 0;
}

```

```
54
55 void do_one_thing(int *pnum_times) {
56   int i, j, x;
57   unsigned long k;
58   int work;
59   for (i = 0; i < 50; i++) {
60     pthread_mutex_lock(&mut);
61     printf("doing one thing\n");
62     work = *pnum_times;
63     printf("counter = %d\n", work);
64     work++; /* increment, but not write */
65     for (k = 0; k < 500000; k++)
66       ;                 /* long cycle */
67     *pnum_times = work; /* write back */
68     pthread_mutex_unlock(&mut);
69   }
70 }
71
72 void do_another_thing(int *pnum_times) {
73   int i, j, x;
74   unsigned long k;
75   int work;
76   for (i = 0; i < 50; i++) {
77     pthread_mutex_lock(&mut);
78     printf("doing another thing\n");
79     work = *pnum_times;
80     printf("counter = %d\n", work);
81     work++; /* increment, but not write */
82     for (k = 0; k < 500000; k++)
83       ;                 /* long cycle */
84     *pnum_times = work; /* write back */
85     pthread_mutex_unlock(&mut);
86   }
87 }
88
89 void do_wrap_up(int counter) {
90   int total;
91   printf("All done, counter = %d\n", counter);
92 }
```

Задание 2

Написать программу для параллельного вычисления факториала по модулю mod (k!), которая будет принимать на вход следующие параметры (пример: -k 10 --pnum=4 --mod=10):

1. k - число, факториал которого необходимо вычислить.
2. pnum - количество потоков.
3. mod - модуль факториала

Для синхронизации результатов необходимо использовать мьютексы.

```
~/oslab2019$ cd lab5/src
~/.../lab5/src$ ./factorial -k 10 --pnum=4 --mod=10
Result: 0
```

```
~/.../lab5/src$ ./factorial -k 10 --pnum=4 --mod=33
Result: 21
~/.../lab5/src$ ./factorial -k 10 --pnum=4 --mod=11
Result: 10
```

```c
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <getopt.h>

struct Args{
  int *result;
  int begin;
  int end;
  int mod;
};

pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;

void ThreadFactorial(void* args){
  struct Args *factorial_args = (struct Args *)args;

  for (uint32_t i = factorial_args->begin; i < factorial_args->end; i++){
    pthread_mutex_lock(&mut);
    *(factorial_args->result) = (*(factorial_args->result)*i) % factorial_args->mod;
    pthread_mutex_unlock(&mut);
  }
}

int main(int argc, char **argv) {
  uint32_t k = -1;
  uint32_t pnum = -1;
  uint32_t mod = -1;

  while (1) {
    int current_optind = optind ? optind : 1;

```

```c
static struct option options[] = {{"pnum", required_argument, 0, 0},
                                  {"mod", required_argument, 0, 0},
                                  {0, 0, 0, 0}};

int option_index = 0;
int c = getopt_long(argc, argv, "k:", options, &option_index);

if (c == -1) break;

switch (c) {
  case 0:
    switch (option_index) {
      case 0:
        pnum = atoi(optarg);
        if (pnum <= 0) {
            printf("pnum must be at least 1.\n");
            exit(1);
        }
        break;
      case 1:
        mod = atoi(optarg);
        if (mod <= 0) {
            printf("mod must be at least 1.\n");
            exit(1);
        }
        break;
      defalut:
        printf("Index %d is out of options\n", option_index);
    }
    break;

  case 'k':
    k = atoi(optarg);
    if (k < 0) {
      printf("k must be at least 0.\n");
      exit(1);
    }
    break;

  case '?':
    break;

  default:
    printf("getopt returned character code 0%o?\n", c);
  }
}
if (optind < argc) {
  printf("Has at least one no option argument\n");
  exit(1);
}
```

```
84
85  if (k == -1 || pnum == -1 || mod == -1) {
86    printf("Usage: %s -k \"num\" --pnum \"num\" --mod \"num\" \n", argv[0]);
87    exit(1);
88  }
89
90  struct Args args[pnum]; // аргументы для тредов
91  pthread_t threads[pnum]; // тут содержатся айдишники тредов
92  int factorial = 1;
93
94  for(uint32_t i = 0; i < pnum; i++)
95  {
96    args[i].result = &factorial;
97    args[i].mod = mod;
98    args[i].begin = i*k/pnum + 1;
99    args[i].end = i == k - 1 ? k + 1 : (i + 1)*k/pnum + 1;
100
101   if (pthread_create(&threads[i], NULL, ThreadFactorial, (void *)&args[i])) {
102     printf("Error: pthread_create failed!\n");
103     return 1;
104   }
105 }
106
107 for (uint32_t i = 0; i < pnum; i++)
108   pthread_join(threads[i], NULL);
109
110 printf("Result: %d\n", factorial);
111 return 0;
112 }
```

Задание 3

Напишите программу для демонстрации состояния deadlock.

Deadlock. Дедлок (тупик) – **состояние** взаимной блокировки двух или более потоков. Такая ситуация возникает в том случае, когда первый поток ждёт освобождения ресурса, которым владеет второй поток, а второй поток ждёт освобождения ресурса, которым владеет первый. Понятно, что в таком **состоянии** могут находиться не только два потока, а несколько.

```
~/.../lab5/src$ gcc -Wall -pthread deadlock.c -o deadlock
~/.../lab5/src$ ./deadlock
The first process locked the mut_1
The first process locked the mut_2
The first process unlocked the mut_2
The first process unlocked the mut_1
The second process locked the mut_2
The second process locked the mut_1
The second process unlocked the mut_1
The second process unlocked the mut_2
~/.../lab5/src$ gcc -Wall -pthread deadlock.c -o deadlock
~/.../lab5/src$ ./deadlock
The first process locked the mut_1
The second process locked the mut_2
```

Сначала было 10 мс, потом 100 мс

```c
#include <errno.h>
#include <unistd.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void do_one_thing(int *);
void do_another_thing(int *);
int common = 0;
pthread_mutex_t mut_1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mut_2 = PTHREAD_MUTEX_INITIALIZER;

int main() {
  pthread_t thread1, thread2;

  if (pthread_create(&thread1, NULL, (void *)do_one_thing,
      (void *)&common) != 0) {
    perror("pthread_create");
    exit(1);
  }

  if (pthread_create(&thread2, NULL, (void *)do_another_thing,
                     (void *)&common) != 0) {
    perror("pthread_create");
    exit(1);
  }

  if (pthread_join(thread1, NULL) != 0) {
    perror("pthread_join");
    exit(1);
  }

  if (pthread_join(thread2, NULL) != 0) {
    perror("pthread_join");
    exit(1);
  }

  return 0;
}

void do_one_thing(int *pnum_times) {
  pthread_mutex_lock(&mut_1);
  printf("The first process locked the mut_1\n");
  usleep(100);
  pthread_mutex_lock(&mut_2);
  printf("The first process locked the mut_2\n");

  pthread_mutex_unlock(&mut_2);
  printf("The first process unlocked the mut_2\n");
  pthread_mutex_unlock(&mut_1);
  printf("The first process unlocked the mut_1\n");
}

void do_another_thing(int *pnum_times) {
  pthread_mutex_lock(&mut_2);
  printf("The second process locked the mut_2\n");
  pthread_mutex_lock(&mut_1);
  printf("The second process locked the mut_1\n");

  pthread_mutex_unlock(&mut_1);
  printf("The second process unlocked the mut_1\n");
  pthread_mutex_unlock(&mut_2);
  printf("The second process unlocked the mut_2\n");
}
```