Операционные системы и сети

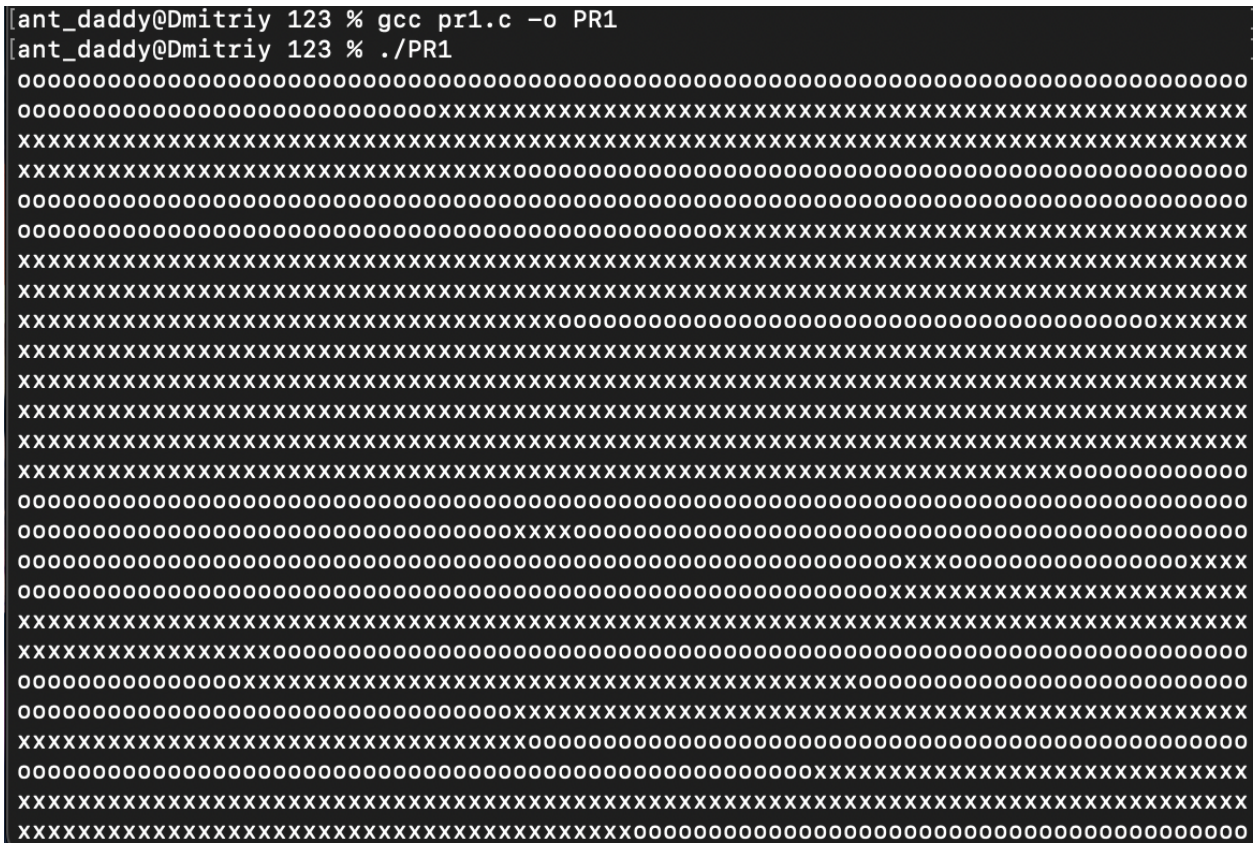
Лабораторная работа №5 Джугели Дмитрий ПИН-31Д

Потоки в ОС Linux

Цель работы: знакомство с системными вызовами для управления потоками в ОС Linux.

Задание 1. Выполните программу **pr1.c**. Программа создает поток, который непрерывно печатает '**x**' на устройстве **stderr**. После создания потока главный поток непрерывно печатает **'o'** на **stderr**. (Приостановить выполнение программы можно при помощи **Ctrl+s**; возобновить - любой клавишей. Снять программу можно при помощи **Ctrl+c**.)

```
/* pr1.c */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
void* print_xs (void* unused)
{ while (1)
  fputc ('x', stderr);
  return NULL;
}
int main ()
{ int p;
  pthread_t thread_id;
  p = pthread_create (&thread_id, NULL, print_xs, NULL);
  if (p != 0) { perror("Thread problem"); exit(1);}
  while (1)
  fputc ('o', stderr);
  return 0;
}
```



*Поясните в отчете: 1) Добавьте в программу печать ID обоих потоков, убрав операторы **while**. Воспользуйтесь функцией **pthread_self**. 2) При помощи команды **ps** выясните и запишите в отчете ID созданных потоков. Сколько потоков создано в программе?*

pr1.c

```c
/* pr1.c */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void* print_xs(void* unused) {
    printf("Thread ID for 'x': %lu\n", (unsigned long)pthread_self());
    while (1)
        fputc('x', stderr);
    return NULL;
}

int main() {
    int p;
    pthread_t thread_id;
    printf("Main Thread ID: %lu\n", (unsigned long)pthread_self());

    // Создание первого потока
    p = pthread_create(&thread_id, NULL, print_xs, NULL);
    if (p != 0) {
        perror("Thread problem");
        exit(1);
    }

    // Печать ID созданного потока
    printf("Thread ID for 'x': %lu\n", (unsigned long)thread_id);

    // Ожидание завершения первого потока
    pthread_join(thread_id, NULL);

    // Создание второго потока
    p = pthread_create(&thread_id, NULL, print_xs, NULL);
    if (p != 0) {
        perror("Thread problem");
        exit(1);
    }

    // Печать ID второго созданного потока
    printf("Thread ID for 'x': %lu\n", (unsigned long)thread_id);

    // Ожидание завершения второго потока
    pthread_join(thread_id, NULL);

    return 0;
}
```

```
[ant_daddy@Dmitriy 123 % gcc pr1.c -o PR1
[ant_daddy@Dmitriy 123 % ./PR1
 Main Thread ID: 140704704786688
 Thread ID for 'x': 123145310683136
 Thread ID for 'x': 123145310683136
 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Передача данных в поток

Т.к. тип аргумента, передаваемого в поток, - **void*,** то для передачи одного параметра типа **int** его следует преобразовать к типу **(void*)**. Для передачи

большего количества параметров аргумент потока должен быть указателем на структуру или область данных, содержащую передаваемые параметры.

(*) Необходимо, чтобы данные, передаваемые новому потоку, были доступны потоку, при этом не следует передавать стековые переменные.

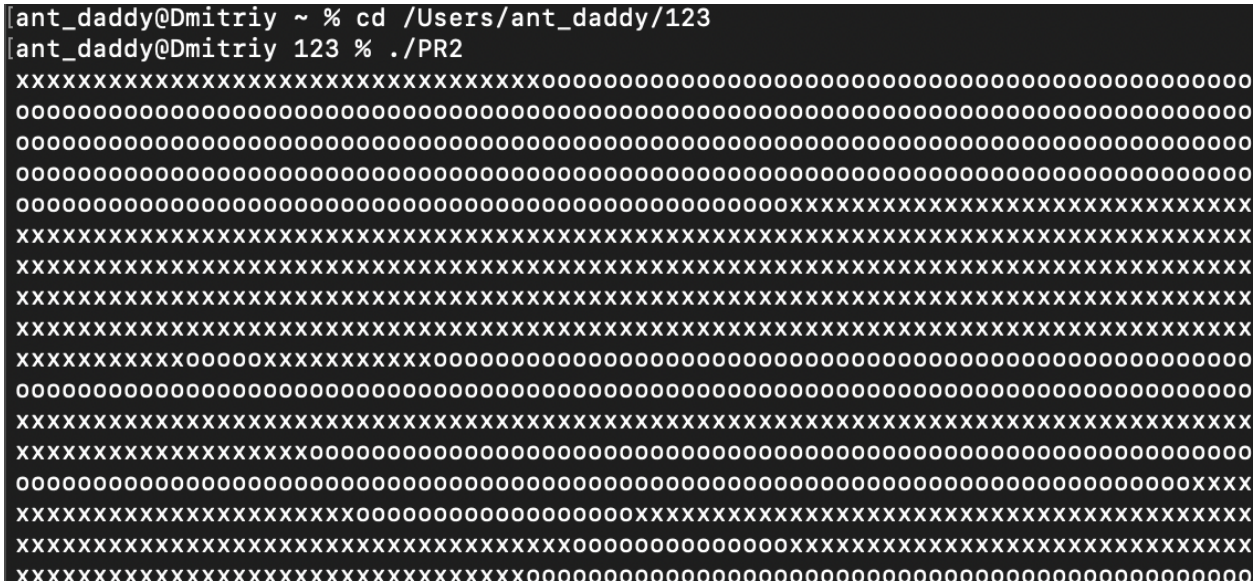
Задание 2. Выполните программу **pr2.c**. Программа создает два новых потока: один печатает **'x'**, другой **'o'** на устройстве **stderr**. Каждый поток печатает определенное количество символов и затем завершается возвратом из функции потока. Оба потока используют одну и ту же функцию, **char_print**, но вызывают ее с разными значениями параметров.

Внимание! Программа содержит ошибку (см. (*)). *Исправьте ошибку и поясните ее причину*.

```
thread1_args.character = 'x';
thread1_args.count = 30000;     /* Печатать 'x' 30000 раз */
pthread_create (&thread1_id, NULL, &char_print, &thread1_args);
thread2_args.character = 'o';
```

Ошибка в неправльных ковычках

Также ошибка в работе указателей на thread1_args и thread2_args




```
/* pr2.c */

#include <pthread.h>

#include <stdio.h>
```

```
struct char_print_parms
{     char character;      /* Символ, который печатать */
      int count;           /* Сколько раз печатать символ */
};
void* char_print (void* parameters)
{     /* Преобразовать указатель к нужному типу */
      struct char_print_parms* p = (struct char_print_parms*)
parameters;
      int i;
      for (i = 0; i < p->count; ++i)
      fputc (p->character, stderr);
      return NULL;
}
int main ()
{
pthread_t thread1_id;
pthread_t thread2_id;
struct char_print_parms thread1_args;
struct char_print_parms thread2_args;
thread1_args.character = 'x';
thread1_args.count = 30000;   /* Печатать 'x' 30000 раз */
pthread_create (&thread1_id, NULL, &char_print, &thread1_args);
thread2_args.character = 'o';
thread2_args.count = 20000; /* Печатать 'o' 20000 раз */
pthread_create (&thread2_id, NULL, &char_print, &thread2_args);
return 0;
}
```

Объединение потоков и возврат значения из потока

В случае потоков аналогом функции **wait** является функция **pthread_join**: поток, вызвавший эту функцию, будет ожидать завершения указанного потока.

Функция возвращает ноль в случае нормального выполнения, и не ноль в случае ошибки. Функция имеет два параметра: ID потока, завершения которого следует ожидать, и переменную типа указатель на **void**, куда будет записано значение, возвращаемое потоком. Если это значение не требуется, то второй параметр функции **pthread_join** может быть NULL.

Задание 3. Создайте программу **pr3.c**, модифицировав программу **pr2.c**: а) добавьте в главный поток вызов функций **pthread_join** для ожидания завершения обоих дочерних потоков; б) верните из дочерних потоков ID потоков и распечатайте их в главном потоке.

pr3.c

```c
/* pr3.c */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

struct char_print_parms
{
    char character; /* Символ, который печатать */
    int count; /* Сколько раз печатать символ */
    pthread_t thread_id; /* ID потока */
};

void* char_print(void* parameters)
{
    struct char_print_parms* p = (struct char_print_parms*)parameters;
    int i;
    for (i = 0; i < p->count; ++i)
        fputc(p->character, stderr);

    return (void*)pthread_self(); /* Возвращаем ID потока */
}

int main()
{
    pthread_t thread1_id;
    pthread_t thread2_id;

    struct char_print_parms *thread1_args = (struct char_print_parms*)malloc(sizeof(struct
char_print_parms));
    struct char_print_parms *thread2_args = (struct char_print_parms*)malloc(sizeof(struct
char_print_parms));

    thread1_args->character = 'x';
    thread1_args->count = 30000; /* Печатать 'x' 30000 раз */
    pthread_create(&thread1_id, NULL, &char_print, thread1_args);
    thread1_args->thread_id = thread1_id;

    thread2_args->character = 'o';
    thread2_args->count = 20000; /* Печатать 'o' 20000 раз */
    pthread_create(&thread2_id, NULL, &char_print, thread2_args);
    thread2_args->thread_id = thread2_id;

    pthread_t *result1;
    pthread_t *result2;
    pthread_join(thread1_id, (void**)&result1);
    pthread_join(thread2_id, (void**)&result2);

    printf("ID потока 1: %p\n", result1);
    printf("ID потока 2: %p\n", result2);

    free(thread1_args);
    free(thread2_args);

    return 0;
}
```

```
[ant_daddy@Dmitriy 123 % ./PR3
 xxxxxxxxxxxxxxxxxxxxxooooooooooooo
 ooooooooooooooooooooooooooooooxxxx
 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
ID потока 1: 0x7000014b9000
ID потока 2: 0x70000153c000
```

Мьютексы в потоках

Если несколько потоков модифицируют значение общей переменной, то возникает состояние гонки. Решение проблемы заключается в том, чтобы в каждый момент времени доступ к такому участку программы (так называемой ***критической секции***, ***CR***) имел только один поток. Это можно сделать при помощи ***мьютексов*** (*mutex*, сокращение от *MUTual Exclusion*).

Мьютекс - это специальный вид семафора, блокировка, которую в каждый момент времени может устанавливать один поток. Чтобы создать мьютекс, нужно объявить переменную типа **pthread_mutex_t** и присвоить ей начальное значение PTHREAD_MUTEX_INITIALIZER. Перед началом CR следует вызвать функцию **pthread_mutex_lock()**, передав ей параметр - указатель на мьютекс. В конце CR следует вызвать функцию **pthread_mutex_unlock()**, передав ей параметр - указатель на мьютекс.

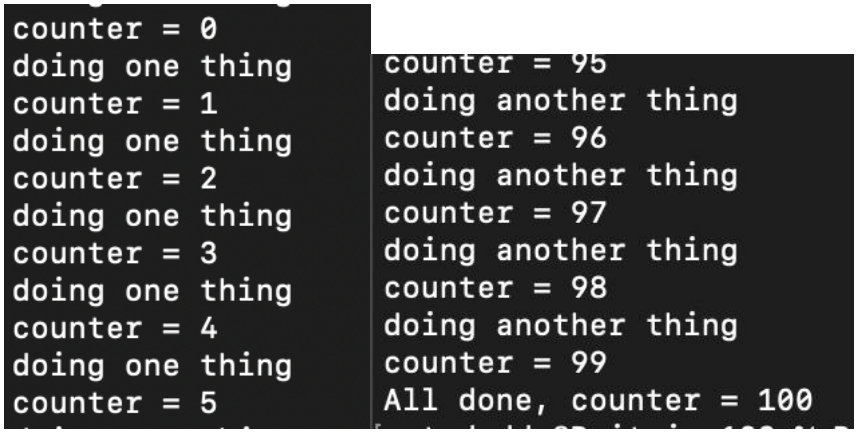
Если мьютекс свободен, т.е. ни один из потоков не находится в CR, то при выполнении функции **pthread_mutex_lock()** мьютекс переходит во владение данного потока и из функции **pthread_mutex_lock()** происходит немедленный возврат. Если же мьютекс уже был захвачен другим потоком, то выполнение функции **pthread_mutex_lock()** блокируется и возобновляется только тогда, когда мьютекс вновь становится свободным. Сразу несколько потоков могут ожидать освобождения мьютекса. Когда это событие наступает, только один из потоков (выбираемый произвольно) разблокируется и получает возможность захватить мьютекс. Остальные потоки остаются заблокированными. Функция **pthread_mutex_unlock()** освобождает мьютекс. Она должна вызываться только из того потока, который захватил мьютекс.

Задание 4.

4.1. Изучите работу программы **mutex.c**, приведенную в Приложении. Скопируйте файл **mutex.c** в ваш каталог.

```
[ant_daddy@Dmitriy 123 % gcc -g mutex.c -o mutex
[ant_daddy@Dmitriy 123 % ./mutex
doing one thing
counter = 0
doing one thing
counter = 1
doing one thing
counter = 2
doing one thing
counter = 3
doing one thing
counter = 4
doing one thing
counter = 5
doing one thing
counter = 6
doing one thing
counter = 7
doing one thing
counter = 8
doing one thing
counter = 9
doing one thing
counter = 10
doing one thing
counter = 11
doing one thing
counter = 12
doing one thing
counter = 13
doing one thing
counter = 14
doing one thing
counter = 15
doing one thing
counter = 16
doing one thing
counter = 17
doing one thing
counter = 18
doing one thing
counter = 19
doing one thing
counter = 20
doing one thing
counter = 21
doing one thing
```

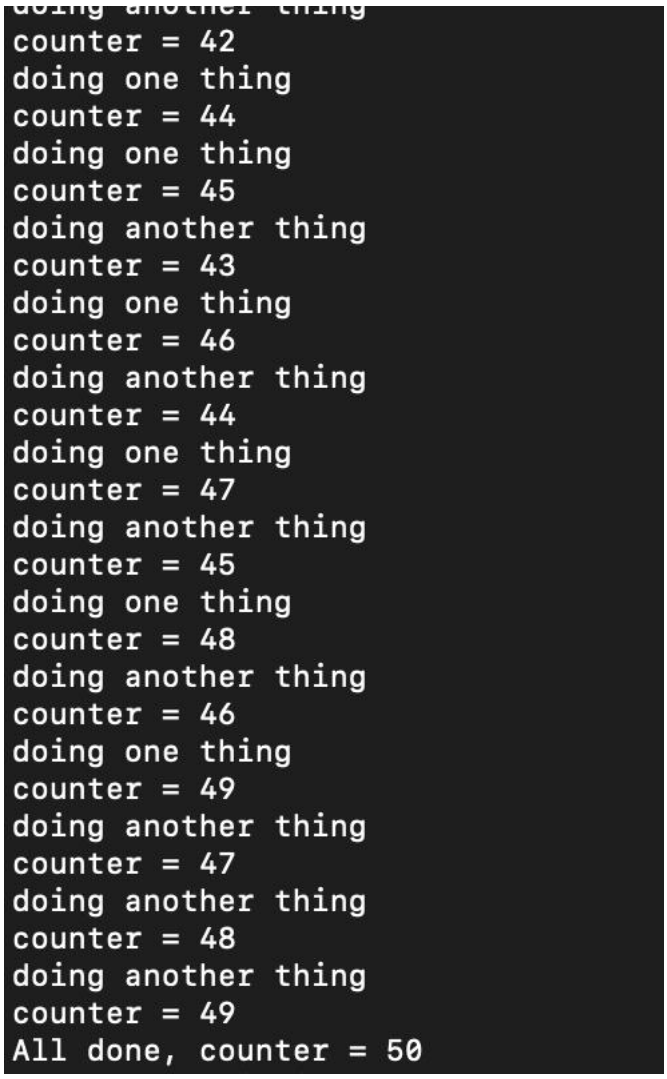
4.2. Выполните программу **mutex**. Проверьте, что переменная **common** (изменяемая параллельно двумя потоками), изменяет свою величину от 0 до 100 (т.к. каждый поток изменяет эту переменную ровно 50 раз). Обратите внимание, что в каждый момент времени переменная **common** читается, увеличивается на 1 и затем записывается без прерывания только одним потоком. Это обеспечивается механизмом мьютексов, который используется обоими потоками (см. функции **pthread_mutex_lock()** и **pthread_mutex_unlock()**).



Оба потока корректно увеличивают значение переменной common от 0 до 100, используя мьютексы для синхронизации доступа к ней.

4.3. Затем удалите механизм мьютексов из программы **mutex_c**, перекомпилируйте ее и выполните несколько раз. Одинаковы ли значения **common** в разных запусках? Всегда ли переменная **common** имеет конечное значение, равное 100? *Объясните результаты в отчете.*

```
[ant_daddy@Dmitriy 123 % gcc -g mutex.c -o mutex
[ant_daddy@Dmitriy 123 % ./mutex
doing another thing
counter = 0
doing one thing
counter = 0
doing one thing
counter = 1
doing another thing
counter = 1
doing another thing
counter = 2
doing one thing
counter = 2
doing another thing
counter = 3
doing one thing
counter = 3
doing another thing
counter = 4
doing one thing
counter = 4
doing another thing
counter = 5
doing one thing
counter = 5
doing another thing
counter = 6
doing one thing
counter = 6
doing another thing
counter = 7
doing one thing
counter = 7
doing another thing
counter = 8
doing one thing
counter = 8
doing another thing
counter = 9
doing one thing
counter = 9
doing another thing
counter = 10
doing one thing
counter = 10
```



Из-за гонок данных получаем дублирующиеся значения counter

Из-за того что потоки исполняются и обращаются к common без синхронизации , значения могут меняться

```
[ant_daddy@Dmitriy 123 % gcc -g mutex.c -o mutex
[ant_daddy@Dmitriy 123 % ./mutex
 Final value of common: 100
```