Механизм POSIX-Thread

Для корректной компиляции и запуска программы `hello.c` необходимо сделать 2 шага:

```
gcc hello.c -lpthread -o hello
./hello
```

POSIX-Thread — это парпрог с общей памятью. То есть все процессы физически имеют доступ к одной и той же области памяти.

Параллельно можно создать сколько угодно потоков, работающих независимо друг от друга. Система сама распределит потоки между ядрами.

Синтаксис работы с потоками в C++

Запускается функция

```
  void * func(void * arg)
```

Замечание: В языке Си тип `void*` обозначает указатель на ЛЮБОЙ тип, в частности, например, указывать на любой массив, или на любую структуру, которую вы захотите.

Все такие функции работают параллельно основной программе.

```
#include <pthread.h>

pthread_t t;//элемент структуры потока

phread_create(pthread_t *t,//элемент структуры
                NULL,//атрибуты вызова
                void* (*func) (void*),//указатель на функцию
                void* arg//аргумент искомой функции
                );
```

Вопрос: Что считается завершением работы потока?

Ответ: Возможны 3 случая:

- Функция завершилась, ничего не возвращая (не была вызвана команда `return`)
- Функция вернула какой-нибудь.
- (!) Корректный выход: `pthread_exit(void*);`

Маленький пример:

```
void * func(void * arg)
{
    int a = 5;
    pthread_exit(arg);
    //pthread_exit(NULL);//как возможный вариант
}
```

Функция `pthread_join` дожидается завершения работы потока.

```
    pthread_join( pthread_t  t, void ** st);
```

В переменную типа `void*` равную `st*`, будет положен результат вычисления потока.

Замечание: Все глобальные переменные будут общими **у всех потоков**.

Компилировать нужно программой `gcc` с опцией `-lpthread`

Mutex (mutual-exclusion)

Можно поставить простой опыт: запустить 10000 процессов, чтобы каждый из них увеличивал значение глобальной переменной на 1. При этом полученное значение не всегда будет равно 10000, это возникает потому, что операция увеличения на 1 не является *атомарной*.

Эта ситуация обходится механизмом *критической секции*.

Как это выглядит на псевдокоде:

```
mutex m;
lock(m);
    i++;
unlock(m);
```

Внимание! Это важно!

Критическую секцию (см. выше) необходимо применять в следующих трёх ситуациях:

- чтение переменной из общей памяти (**всегда**, даже когда кажется, что это не нужно!!! даже когда вы заведомо знаете, что другие процессы в неё не будут ничего писать)
- операции инкрементирования (это по сути совмещает чтение и запись)
- запись.

Это тоже важно!

Критическую секцию надо использовать с умом. Все **вычисления** необходимо вести вне критической секции, чтобы не замедлять общую работу программы, а использовать её **только** для чтения или записи общих данных.

Синтаксис работы с Mutual-Exclusion

`pthread_mutex_t` — тип переменной взаимного исключения. Перед тем, как вызывать механизм критической секции, необходимо инициализировать его:

```
pthread_mutex_init(pthread_mutex_t * m, NULL) //инициализация
pthread_mutex_lock(pthread_mutex_t * m)   //блокировка
pthread_mutex_unlock(pthread_mutex_t * m) //разблокировка
pthread_mutex_destroy(pthread_mutex_t * m) // завершение работы
```

Ниже рассмотрены два примера, которые иллюстрируют необходимость использования критических секций (надо скопировать код и запустить на удалённой машине, или на своём линуксе, если есть)

Пример 1. hello.c

Код 1: hello.c

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

#define TNUM 10000

void *func(void * arg)
{
  printf("hello world\n");
  pthread_exit(NULL);
}

int main()
{
  pthread_t t[TNUM];
  int i;
  for (i = 0; i < TNUM; ++i)
  {
    printf("%d", i);
    pthread_create(&t[i], NULL, func, NULL);
  }
  void * st;
  for (i = 0; i < TNUM; ++i)
  {
    pthread_join(t[i], &st);
  }
  return 0;
}
```

Пример 2. phello.c

Код 2: phello.c

```
#include <stdio.h>
#include <pthread.h>

#define TNUM 10000

int num;
pthread_mutex_t m;

void *func(void * arg)
{
  pthread_mutex_lock(&m);
    num++;
  pthread_mutex_unlock(&m);
  pthread_exit(NULL);
}

int main()
{
  pthread_t t[TNUM];
  num = 0;

  pthread_mutex_init(&m, NULL);

  int i;
  for (i = 0; i < TNUM; ++i)
  {
    pthread_create(&t[i], NULL, func, NULL);
  }
  void * st;
  for (i = 0; i < TNUM; ++i)
  {
    pthread_join(t[i], &st);
  }

  printf("%d\n", num);

  pthread_mutex_destroy(&m);
  return 0;
}
```