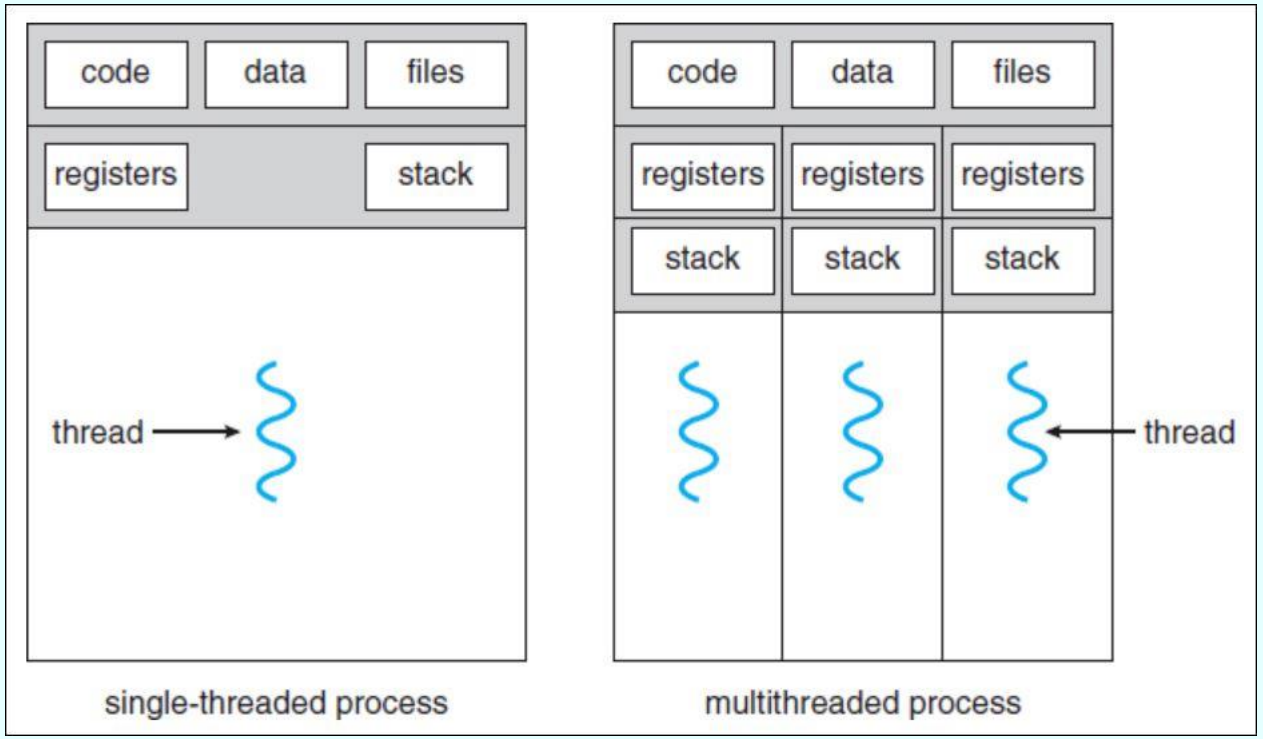
Технологии параллельного программирования на C++

Семинар 3

Понятие о нити исполнения (thread)

- Нити разделяют пользовательский контекст процесса
- Каждая нить исполнения имеет свой стек исполнения, программный счетчик и т.д.
- Каждый процесс имеет хотя бы одну нить исполнения: начальная нить исполнения
- Каждая нить имеет свой уникальный ID в системе: pthread_self

Понятие о нити исполнения (thread)



Функция pthread_self()

```
#include <pthread.h>
pthread_t pthread_self(void);
```

- Возвращает идентификатор текущей нити исполнения

Функция pthread_create()

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
   void * (*start_routine)(void *), void *arg);
```

- Создает новую нить исполнения внутри текущего процесса
- В переменную **thread** сохраняется id нового thread'а
- Параметр **attr** – атрибуты процесса. Будем использовать nullptr.
- **start_routine** – ф-ция, которую будет выполнять новый thread
- Параметр **arg** передается в функцию **start_routine** в качестве аргумента

Возвращает

- **0**, при нормальном завершении
- **Положительное значение**, в случае ошибки

Завершение работы thread’а

- С помощью функции pthread_exit
- С помощью функции pthread_cancel
- С помощью возврата из функции, ассоциированной с нитью исполнения
- Потоковая функция выполняет return
- Если в процессе выполняется возврат из функции main() или в любой нити исполнения осуществляется вызов exit, это приводит к завершению всех thread’ов процесса

Функция pthread_exit()

```
#include <pthread.h>
void pthread_exit(void *status);
```

- Завершает работу thread’а текущего процесса
- Функция никогда не возвращается в вызвавший ее thread
- Объект, на который указывает параметр (**status**) этой функции может быть изучен другой нитью исполнения
- Параметр **status** должен указывать на статическую переменную
- Функция **pthread_exit** освобождает всю память, занятую данными нити, включая стек нити.
- Если главная нить вызывает **pthread_exit**, то остальные нити продолжают исполняться

Функция pthread_join

```
#include <pthread.h>
int pthread_join (pthread_t thread,
void **status_addr);
```

- Блокирует работу вызвавшей ее нити исполнения до завершения thread’а с идентификатором **thread**
- После разблокировки в указатель, расположенный по адресу **status_addr**, заносится адрес, который вернул завершившийся thread. Возвращает
- **0**, при успешном завершении
- **Положительное значение**, в случае ошибки

Функция pthread_cancel

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

- Посылает запрос на завершение потока **thread**
- Возвращает **0**, при успешном завершении
- Поток может выполнить проигнорировать запрос, выполнить его немедленно или отложить выполнение запроса Для определения, какое действие нужно выполнить есть функции:

```
int pthread_setcancelstate(int state, int *oldstate);
```

Запрос игнорируется, если аргументом **state** является **PTHREAD_CANCEL_DISABLE;** разрешено, если **state** имеет значение **PTHREAD_CANCEL_ENABLE**

```
int pthread_setcanceltype(int type, int *oldtype);
```

Если завершение разрешено, вызывается **pthread_setcanceltype.** Завершение выполняется немедленно, если аргумент **type** имеет значение **PTHREAD_CANCEL_ASYNCHRONOUS**. Если **type** имеет значение **PTHREAD_CANCEL_DEFERRED**, запрос на завершение откладывается до следующей точки завершения.

Функция pthread_testcancel

```
#include <pthread.h>
void pthread_testcancel(void);
```

- Проверяет, есть ли ожидающий обработки запрос на завершение
- Завершение выполняется сразу же, если запрос на завершение находится в состоянии ожидания обработки, когда вызывается эта функция.

Пример 1. Программа с 2 thread'ами

```
#include <iostream>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
using namespace std;
int a=0;

void *mythread(void *value)
{  pthread_t mythid;
   mythid = pthread_self();
   a = a+1;
   cout<<"Thread and result= "<<mythid<<" "<<a<<endl;
   return 0;  }
int main()
{  pthread_t thid, mythid;
   int result;
   result = pthread_create(&thid, nullptr, mythread, nullptr);
   if (result !=0){
        cout<<"Error on thread create, return value "<<result<<endl;
        exit(-1); }
   mythid = pthread_self();
   sleep(1);
   a = a + 1;
   cout<<"Thread created "<<thid<<endl;
   cout<<"Thread and result= "<<mythid<<" "<<a<<endl;
   pthread_join(thid, nullptr);
  return 0;}
```

Пример 1

- Что выведет программа?
- Что будет выводить, если убрать функцию sleep ?

Компиляция программы

- С помощью параметра **–pthread** подключаем библиотеку pthread

- Пример:

  g++ test.cpp -pthread

Организация взаимодействия процессов

- Критическая секция – часть программы, исполнение которой может привести к возникновению race condition (состязание за вычисление)

Требования к алгоритмам

- Если процесс Pi исполняется в критическом участке, то не существует других процессов, которые исполняются в соответствующих критических секциях
- Процессы, которые находятся вне своих критических участков, не могут препятствовать другим процессам входить в их собственные критические участки
- Не должно возникать неограниченно долгого ожидания для входа одного из процессов в свой критический участок

Мьютексы

- Мьютекс – это экземпляр типа pthread_mutex_t. Перед использованием необходимо инициализировать мьютекс функцией pthread_mutex_init

```
int pthread_mutex_init(pthread_mutex_t *mutex,const pthread_mutexattr_t *attr);
```

- Если мьютекс создан статически и не имеет дополнительных параметров, то он может быть инициализирован с помощью макроса PTHREAD_MUTEX_INITIALIZER
- После использования мьютекса его необходимо уничтожить с помощью функции

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- После создания мьютекса он может быть захвачен с помощью функции

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- После этого участок кода становится недоступным остальным потокам – их выполнение блокируется до тех пор, пока мьютекс не будет освобождён. Освобождение должен провести поток, заблокировавший мьютекс, вызовом

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Задача 1

- Модифицировать пример 1, добавив третью нить исполнения: увеличение значения «a» на 3

Задача 2

- Модифицируйте пример 1, добавив mutex для корректного вывода результатов

Задача 3

- Нужно создать нить исполнения и передать ей значение типа int. В потоковой функции прибавить к этому значению 1 и распечатать.

Задача 4

- Дан массив размера N. Создать N-1 потоков, каждый из которых будет рассчитывать сумму соседних элементов массива. Вывести идентификатор потока и соответствующее посчитанное значение.

Задача 5

- Найти скалярное произведение двух векторов, используя потоки и мьютексы.