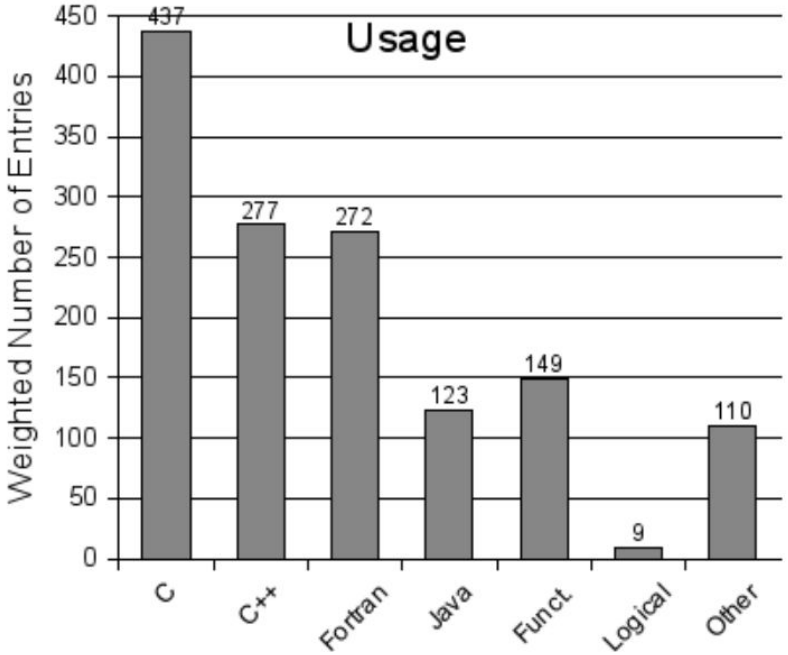
Введение в нити стандарта posix (практика)

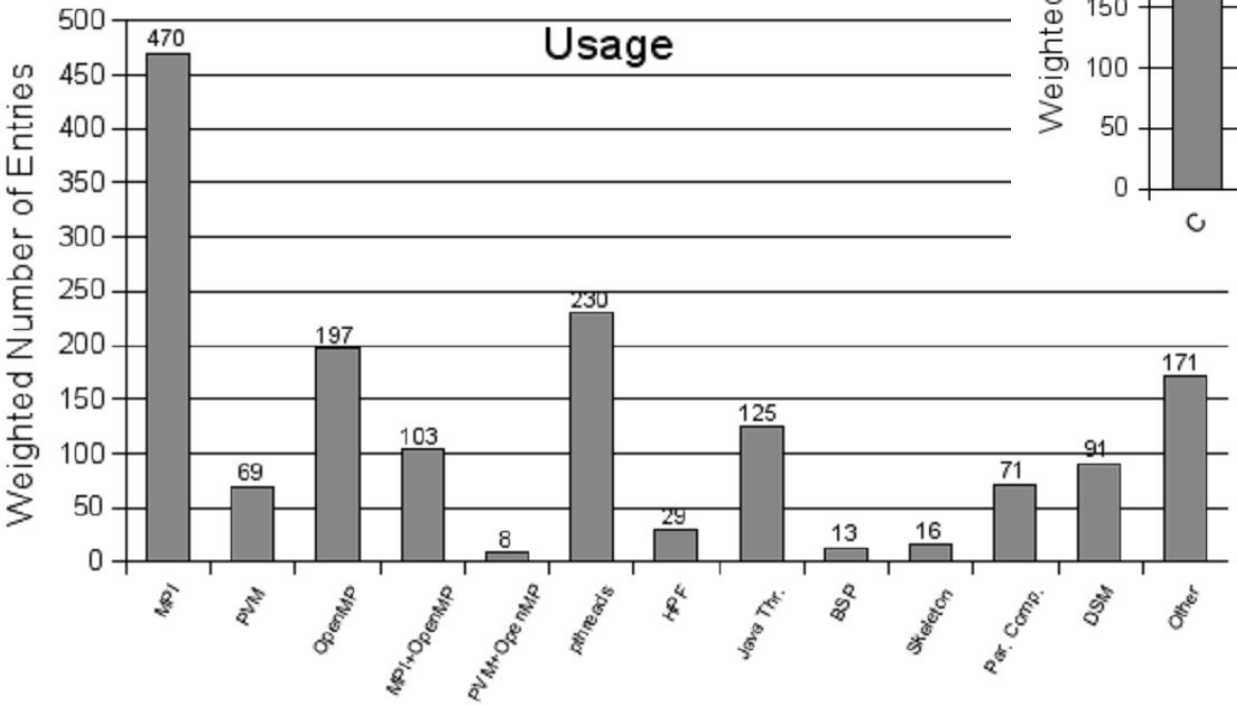
Роль и место параллельных языков и библиотек

* - по данным опроса

Suess M, Leopold C. Observations on the Publicity and Usage of Paral-
lel Programming Systems and Languages: A Survey Approach. (2007)



Наиболее популярные языки параллельного программирования*

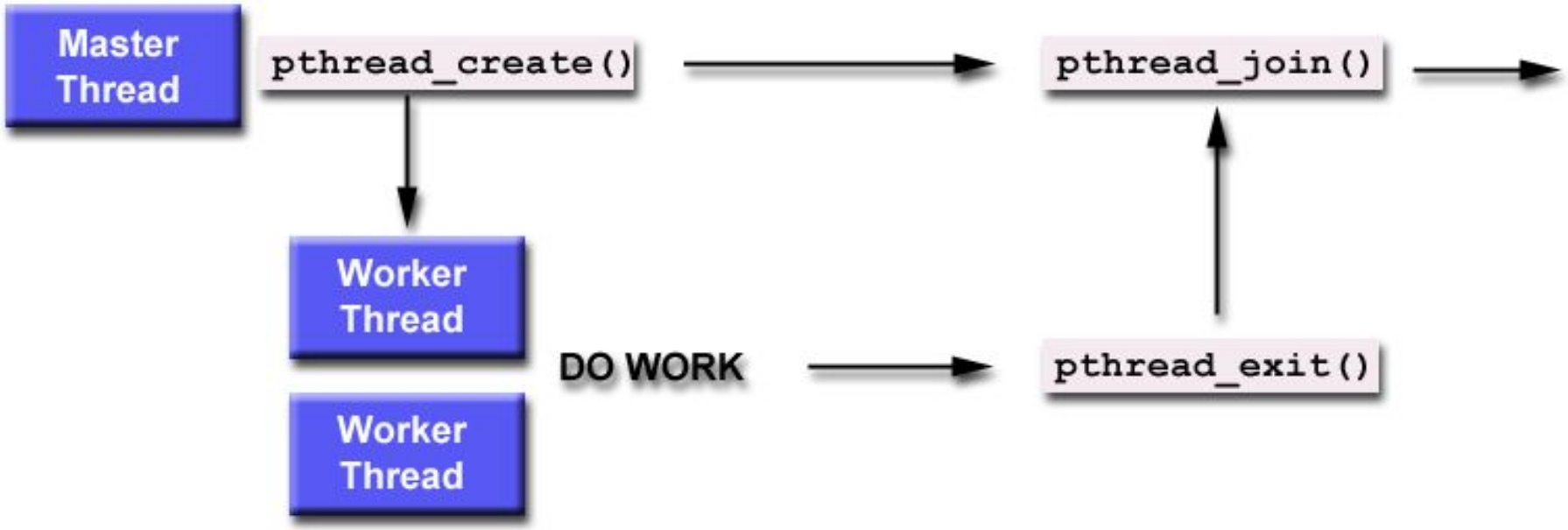


Наиболее популярные библиотеки параллельного программирования*

Программирование с использованием нитей Posix

- Нить (поток) очень похожа на процесс, ее еще называют легковесным процессом.
- Основные отличия от процесса: процессу соответствует своя независимая от других процессов область памяти, таблица открытых файлов, текущая директория и др. информация уровня ядра. У всех нитей, принадлежащих одному процессу все эти сущности общие.
- Стандарт Posix был принят в 1995г и существует на UNIX и Linux-подобных системах (до 1995 г. был только на UNIX-подобных системах). С этим стандартом совместимы или практически совместимы почти все существующие операционные системы: BSD, Mac OS, Solaris и др. Существует реализация соответствующих библиотек для Windows OS.
- Помимо Posix threads, существуют следующие распростаненные реализации программирования для систем с общей памятью (SMP – symmetric multiprocessing): OpenMP, Windows threads.

Схема работы параллельной программы с использованием нитей



Компиляция и запуск

Компиляция:

gcc file_name.c **-lpthread** **-lrt** (по умолчанию исполняемы файл имеет имя «a.out»)

Запуск:

./a.out – как обычно

Первая программа, часть 1 создание нити

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>        //   заголовочный файл pthread

int main(int argc, char *argv[]){
int param;
int rc;
void *arg;
pthread_t pthr;     // тип данных pthread
1) rc = pthread_create(&pthr, NULL, start_func, NULL);   // создание нити  упрощенный вызов
2) rc = pthread_create(&pthr, NULL, start_func, (void*) &param);
```

Первая программа, часть 2

/* нить завершает свою работу, когда происходит выход из функции start_func. Если необходимо получить возвращаемое значение функции, то нужно использовать*/

1) pthread_join(pthr, NULL); // по умолчанию ничего не принимаем

2) pthread_join(pthr, &arg); // записываем по адресу &arg адрес возвращаемой переменной

} // конец функции main

Первая программа, часть 3 функция нити

```
/* Функция работы нити (область жизни нити)*/

void* start_func(void* param){
    int *local;
    local = (int*) param;
    1)  pthread_exit( NULL);        // используется по умолчанию
        2)  pthread_exit(  (void*)  &arg);      // используется, если
необходимо завершить выполнение нити раньше с возвратом
значения, возможно использование в качестве аргумента
NULL , переменная arg должна быть выделена динамически
или иметь класс памяти static
    3) return NULL;    //  или так
}   // в момент выхода из функции start_func нить освобождает свои
ресурсы (погибает)
```

Первая программа, часть 4 семафоры

Механизм синхронизации нитей. Существует несколько механизмов, организующих доступ к критической секции.

Критическая секция – участок кода программы, содержащий общий ресурс, и доступ к которому предоставляется лишь одной нити.

N-мерный семафор принимает значения от 0 до N, мы будем пользоваться только бинарным семафором, принимающим значения 0 и 1

Первая программа, часть 5 семафоры

```
#include<semaphore.h>

sem_t   sem;     // объявление семафора – глобальной переменной
int main(int argc, char *argv[]){
     sem_init (&sem, 0, 1);     // инициализация семафора
     …
     sem_destroy(&sem); // освобождение семафора
     return 0;
}
```

Первая программа, часть 6 критическая секция

```
void* start_func(void* param){
        int val;
        …
        sem_wait(&sem);    // уменьшает значение на 1,
если sem = 0 в момент выполнения функции, то нить ожидает
        …                          // действия над общей переменной
        sem_post(&sem);  // увеличивает значение семафора на 1

        sem_getvalue(&sem,  &val);    // проверка значения семафора
    return NULL;
```

критическая секция

Полезные блоки программы

```
// Создание нескольких нитей
#define NUM_THREADS     2
int main (int argc, char *argv[])
{
    pthread_t pthr[NUM_THREADS];
    void *arg;
    …
    for(i = 0; i < NUM_THREADS; i++){
        rc = pthread_create(&pthr[i], NULL, start_func, NULL);
        if (rc)  printf("ERROR; return code from pthread_create() is %d \n", rc);
    }
    …
    for(i = 0; i < NUM_THREADS; i++){
        rc = pthread_join(pthr[i], &arg);
        printf("value from func  %d  \n", *(int*)arg);      // в качестве
примера
        if (rc)  printf("ERROR; return code from pthread_join() is %d \n", rc);
    }
 }
```

Время работы программы

```
#include<time.h>   // заголовочный файл, содержащий типы и функции для работы с датой и временем

struct timespec begin, end;   // требует использования ключа –lrt  !
double elapsed;

clock_gettime(CLOCK_REALTIME, &begin);    /* возвращает ссылку на
                запись типа timespec, которая
объявлена в time.h с полями
                        time_t tv_sec; –  секунды,
                        long tv_nsec; – наносекунды.
                        */
…  // здесь работают нити

clock_gettime(CLOCK_REALTIME, &end);

elapsed = end.tv_sec - begin.tv_sec;    // время в секундах
elapsed += (end.tv_nsec - begin.tv_nsec) / 1000000000.0;     // добавляем время вплоть до наносекунд
```

О функции генерации случайных чисел 1

Функция rand() не является нитезащищенной. Она имеет следующий примерный вид:

```
unsigned int next = 1;

/* rand - return pseudo-random integer on 0..32767 */
int rand(void) {
   next = next*1103515245 + 12345;
   return (unsigned int)(next/65536) % 32768;
}

/* srand - set seed for rand() */
void srand(unsigned int seed) {
   next = seed;
}
```

и имеет дело с глобальной промежуточной переменной next, которая меняется при каждом вызове каждой из нитей. Это промежуточное значение хранится в статической области памяти, то есть является общей. Помимо неповторяемости результата, это приводит к сильному замедлению программы.

О функции генерации случайных чисел 2

С библиотекой стандарта Posix поставляется аналог ф-ии rand(), но уже нитезащищенный. Его вид:

```
/* rand_r – a reentrant pseudo-random integer on 0..32767 */
int rand_r(unsigned int *nextp) {
   *nextp = *nextp * 1103515245 + 12345;
   return (unsigned int)(*nextp / 65536) % 32768;
}
```

то есть ф-ия имеет дело с локальной переменной нити, которую нужно объявить в функции-обработчике нити. Такая ф-ия называется реентерабельной (от англ. reentrant — повторно входимый). Поскольку промежуточное значение - уже локальная переменная, то конфликта доступа не возникает.

Как это выглядит в программе:

```
int x_k, y_k, z_k;

 x = ((float)rand_r(&x_k) / RAND_MAX) * (i_final - i_init) + i_init;
 y = (float)rand_r(&y_k) / RAND_MAX;
 z = (float)rand_r(&z_k) * Z0 / RAND_MAX;
```