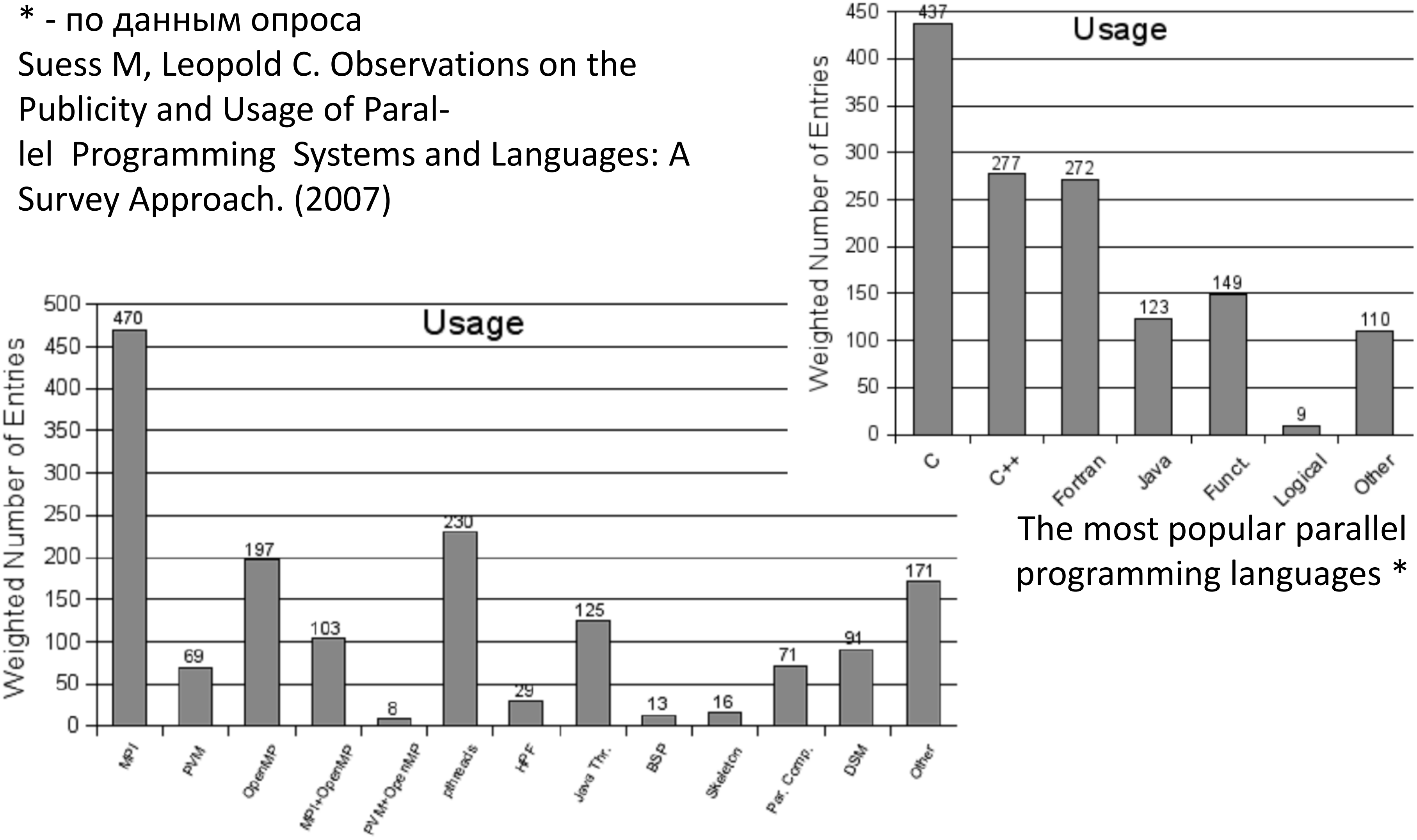
Introduction into the
Posix threads (Pthreads)

Role and place of parallel languages and libraries

* - по данным опроса
Suess M, Leopold C. Observations on the Publicity and Usage of Parallel Programming Systems and Languages: A Survey Approach. (2007)



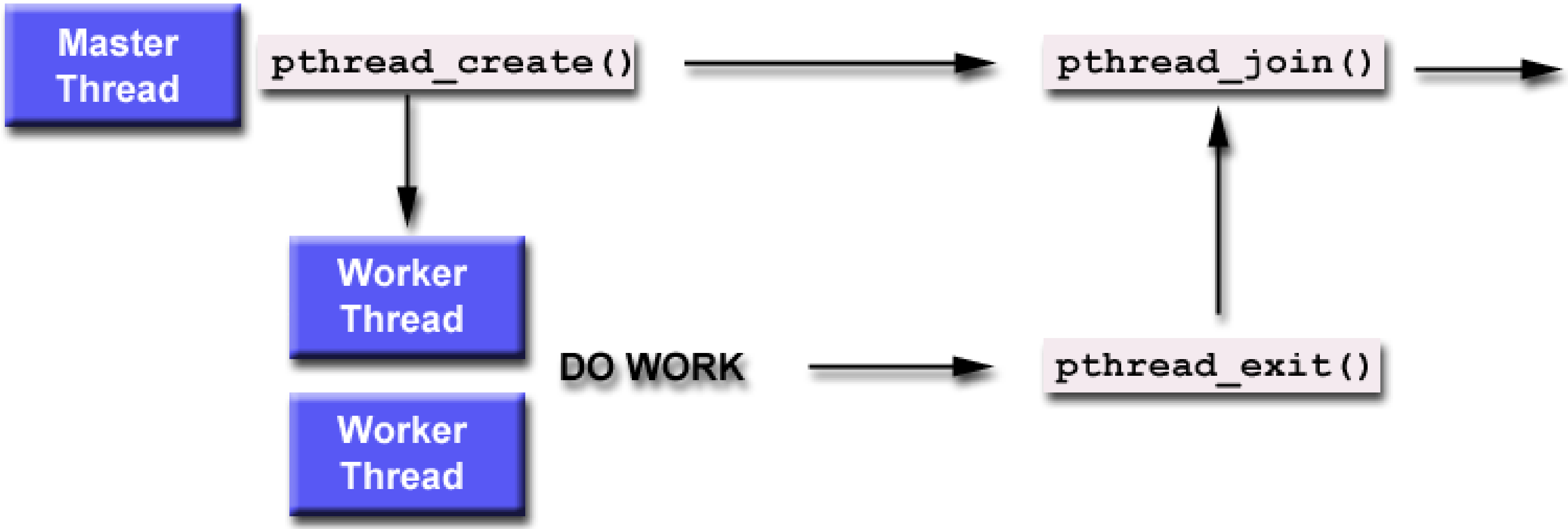
The most popular parallel programming languages *

The most popular parallel programming libraries *

Programming with Posix threads

- Thread is very similar to a process. It is also called as lightweight process.
- The main differences from a process. Process has its own memory area, independent of other processes, table of open files, current directory, and other kernel-level information. All threads, belonging to the same process, have all these entities in common.
- The Posix standard was adopted in 1995 and exists on UNIX and Linux-like systems (until 1995 it was only on UNIX-like systems). Almost all existing operating systems are compatible or practically compatible with this standard: BSD, Mac OS, Solaris, etc. There is an implementation of the corresponding libraries for Windows OS.
- In addition to Posix threads, there are the following common programming implementations for systems with shared memory (SMP - symmetric multiprocessing): OpenMP, Windows threads.

Scheme of a parallel program using Pthreads



Master Thread

pthread_create()

pthread_join()

Worker Thread

Worker Thread

DO WORK

pthread_exit()

Compilation and running

Compilation :

```
gcc     file_name.c   -lpthread   -lrt
```

(by default the executable is named "a.out")

Running:

./a.out – as usual

1st program, part 1
thread creation

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>          //   pthread header file

int main(int argc, char *argv[]){
int param;
int rc;
void *arg;
pthread_t pthr;     // data type pthread
1) rc = pthread_create(&pthr, NULL, start_func, NULL);   // thread creation, simplified call
2) rc = pthread_create(&pthr, NULL, start_func, (void*) &param);
```

1st program, part 2

```
    /* thread finishes when it exits the start_func function. If you need to get the return value of a function, then you need to use */

    1) pthread_join(pthr,  NULL); // accept nothing by default
    2) pthread_join(pthr, &arg);   // write the address of the returned variable into &arg

}     // end of the main function
```

1st program, part 3
thread’s function

```
/* Thread’s work function (thread’s life area) */

void* start_func(void* param){
    int *local;
    local = (int*) param;
    1)   pthread_exit( NULL);        // used by default
    2) pthread_exit( (void*) &arg);     // it is used if it is necessary to
terminate the execution of a thread earlier with a return value. It is
possible to use NULL as an argument, the arg variable must be
dynamically allocated or have a static memory class
    3) return NULL;     //  or like this
}    // at the moment of exiting the start_func function, thread releases its
resources (dies)
```

1st program, part 4 semaphores

Thread synchronization mechanism. There are several mechanisms for organizing access to the critical section.

Critical section is a piece of program code that contains a shared resource and is only accessible to one thread.

An N-dimensional semaphore takes values from 0 to N, we will only use a binary semaphore that takes values 0 and 1

1st program, part 5
semaphores

```
#include<semaphore.h>

sem_t   sem;     // semaphore declaration – as a global variable

int main(int argc, char *argv[]){
     sem_init (&sem, 0, 1);     // semaphore initialization
```

attribute   initial value

```
    …
    sem_destroy(&sem); // releasing of the semaphore
    return 0;
}
```

1st program, part 6
semaphores

Critical section

```
void* start_func(void* param){
        int val;
        …
        sem_wait(&sem);    // decreases the semaphore value by 1
                          if sem = 0 at the time of this function
                          execution, then thread waits          …
                          // actions on a shared variable
         sem_post(&sem);  // increases the semaphore value by 1

     sem_getvalue(&sem, &val);  // checking the semaphore value
     return NULL;
}
```

Some software design

```
// several threads creation
#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);      // example
        if (rc)  printf("ERROR; return code from pthread_join() is %d \n", rc);
    }
}
```

Determining the running time of a parallel program

```
 #include<time.h>   // header file containing types and functions for working
with date and time

struct timespec begin, end;   // requires  key   –lrt  !
double elapsed;

clock_gettime(CLOCK_REALTIME, &begin);    /* returns a reference to a record
                                        of type timespec, which is declared in
                                        time.h and has fields:
                                        time_t tv_sec; –  seconds,
                                        long tv_nsec; – nanoseconds.
                                        */
…  // threads work here

clock_gettime(CLOCK_REALTIME, &end);

elapsed = end.tv_sec - begin.tv_sec;    // time in seconds
elapsed += (end.tv_nsec - begin.tv_nsec) / 1000000000.0;     // add time down
to nanoseconds
```

On the function of generating random numbers 1

The rand () function is not thread-safe. It looks like this:

```
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;
}
```

and deals with the global intermediate variable next, which changes every time each thread is called. This intermediate value is stored in a static memory area, that is, it is shared. In addition to the non-repeatability of the result, this leads to a strong slowdown in the program.

On the function of generating random numbers 2

An analogue of the rand () function is supplied with the Posix standard library, but already thread-safe. Its appearance:

```
/* 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;
}
```

that is, the function deals with a thread local variable to be declared in the thread handler function. Such a function is called reentrant. Since the intermediate value is already a local variable, there is no access conflict.

In a program:

```
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;
```