

Operating Systems

Laboratory 7

Threads

❑ Threads:

❑ Passing multiple arguments using **structure**


Syntax:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), (void *) & thread_Structure);
```

4


 Passing a structure

```
struct thread_parms {  
    int N;  
    int start;  
    int stop;  
};
```




```
void* runner(void* param) {
```

```
    struct thread_parms* p = (struct thread_parms*) param;  
    /* tests all divisors of the assigned interval */  
    ...  
    ...  
    ...  
}
```



```
int main() {
```

```
    struct thread_parms thread_data;  
    err = pthread_create(&tid, NULL, &thread Function, (void*)& thread_data);  
    ...  
    ...  
    ...  
}
```



Genome= CACCATAAAACCGTGTGGGA

Sequence= CCGT

Length of Genome = 20

Length of Sequence = 4

Number of threads = 4

Steps = ?

```
int main() {

    printf("Insert the genome filename:");
    //Recieve the genome file name from the user

    printf("Insert the sequence filename:");
    //Recieve the sequence file name from the user

    // read the files and save them in a string
    fgenome = fopen(genome_filename, "r");
    // read the files and save them in a string
    fseq = fopen(seq_filename, "r");

    printf("Insert the number of threads:");
    //Recieve the number of threads from the user

    /* Allocate an array to store the parms of all created threads*/

    // Define ID of threads
    pthread_t ....
    // Define struct of each thread
    struct thread_parms ...

    for (i = 0; i < number of threads; i++) {

        /* Prepare the thread parameters */
        //The start point where thread should start to search for the sequence = start

        // The End point where thread should stop to search for sequence= Stop
        Stop = start + Step;

        //Create the thread
        err = pthread_create(&thread ID, NULL, &thread Fucntion, (void*)& thread_struct);
        start += step;
    }
    /* wait for all thread to finish */
    for (i = 0; i < number of threads; i++) {
        pthread_join(thread ID, NULL);
    }

}
```

Genome= CACCATAAAACCGTGTGGGA

Sequence= CCGT

Length of Genome = 20

Length of Sequence = 4

Number of threads = 4

Steps = ?

Steps = number of tests / number of threads

Number of tests = Length of Genome – Length of Sequence

Test 1

Genome= CACCATAAAACCGTGTGGGA

```
int main() {

    printf("Insert the genome filename:");
    //Recieve the genome file name from the user

    printf("Insert the sequence filename:");
    //Recieve the sequence file name from the user

    // read the files and save them in a string
    fgenome = fopen(genome_filename, "r");
    // read the files and save them in a string
    fseq = fopen(seq_filename, "r");

    printf("Insert the number of threads:");
    //Recieve the number of threads from the user

    /* Allocate an array to store the parms of all created threads*/

    // Define ID of threads
    pthread_t ....
    // Define struct of each thread
    struct thread_parms ...

    for (i = 0; i < number of threads; i++) {

        /* Prepare the thread parameters */
        //The start point where thread should start to search for the sequence = start

        // The End point where thread should stop to search for sequence= Stop
        Stop = start + Step;

        //Create the thread
        err = pthread_create(&thread ID, NULL, &thread Fucntion, (void*)& thread_struct);
        start += step;
    }
    /* wait for all thread to finish */
    for (i = 0; i < number of threads; i++) {
        pthread_join(thread ID, NULL);
    }

}
```

Genome= CACCATAAAACCGTGTGGGA

Sequence= CCGT

Length of Genome = 20

Length of Sequence = 4

Number of threads = 4

Steps = ?

Steps = number of tests / number of threads

Number of tests = Length of Genome – Length of Sequence

Test 2

Genome= CACCATAAAACCGTGTGGGA

```
int main() {

    printf("Insert the genome filename:");
    //Recieve the genome file name from the user

    printf("Insert the sequence filename:");
    //Recieve the sequence file name from the user

    // read the files and save them in a string
    fgenome = fopen(genome_filename, "r");
    // read the files and save them in a string
    fseq = fopen(seq_filename, "r");

    printf("Insert the number of threads:");
    //Recieve the number of threads from the user

    /* Allocate an array to store the parms of all created threads*/

    // Define ID of threads
    pthread_t ....
    // Define struct of each thread
    struct thread_parms ...

    for (i = 0; i < number of threads; i++) {

        /* Prepare the thread parameters */
        //The start point where thread should start to search for the sequence = start

        // The End point where thread should stop to search for sequence= Stop
        Stop = start + Step;

        //Create the thread
        err = pthread_create(&thread ID, NULL, &thread Fucntion, (void*)& thread_struct);
        start += step;
    }
    /* wait for all thread to finish */
    for (i = 0; i < number of threads; i++) {
        pthread_join(thread ID, NULL);
    }

}
```


Genome= CACCATAAAACCGTGTGGGA

Sequence= CCGT

Length of Genome = 20

Length of Sequence = 4

Number of threads = 4

Steps = ?

Steps = number of tests / number of threads

Number of tests = Length of Genome – Length of Sequence

Test 3

Genome= CACCAT^{Test 3}AAAACCGTGTGGGA

```
int main() {

    printf("Insert the genome filename:");
    //Recieve the genome file name from the user

    printf("Insert the sequence filename:");
    //Recieve the sequence file name from the user

    // read the files and save them in a string
    fgenome = fopen(genome_filename, "r");
    // read the files and save them in a string
    fseq = fopen(seq_filename, "r");

    printf("Insert the number of threads:");
    //Recieve the number of threads from the user

    /* Allocate an array to store the parms of all created threads*/

    // Define ID of threads
    pthread_t ....
    // Define struct of each thread
    struct thread_parms ...

    for (i = 0; i < number of threads; i++) {

        /* Prepare the thread parameters */
        //The start point where thread should start to search for the sequence = start

        // The End point where thread should stop to search for sequence= Stop
        Stop = start + Step;

        //Create the thread
        err = pthread_create(&thread ID, NULL, &thread Fucntion, (void*)& thread_struct);
        start += step;
    }
    /* wait for all thread to finish */
    for (i = 0; i < number of threads; i++) {
        pthread_join(thread ID, NULL);
    }

}
```

Genome= CACCATAAAACCGTGTGGGA

Sequence= CCGT

Length of Genome = 20

Length of Sequence = 4

Number of threads = 4

Steps = ?

Steps = number of tests / number of threads

Number of tests = Length of Genome – Length of Sequence

Test 3 ... Test L of genome – L of sequence

Genome= CACCATAAAACCGTGTGGGA

```
int main() {

    printf("Insert the genome filename:");
    //Recieve the genome file name from the user

    printf("Insert the sequence filename:");
    //Recieve the sequence file name from the user

    // read the files and save them in a string
    fgenome = fopen(genome_filename, "r");
    // read the files and save them in a string
    fseq = fopen(seq_filename, "r");

    printf("Insert the number of threads:");
    //Recieve the number of threads from the user

    /* Allocate an array to store the parms of all created threads*/

    // Define ID of threads
    pthread_t ....
    // Define struct of each thread
    struct thread_parms ...

    for (i = 0; i < number of threads; i++) {

        /* Prepare the thread parameters */
        //The start point where thread should start to search for the sequence = start

        // The End point where thread should stop to search for sequence= Stop
        Stop = start + Step;

        //Create the thread
        err = pthread_create(&thread ID, NULL, &thread Fucntion, (void*)& thread_struct);
        start += step;
    }
    /* wait for all thread to finish */
    for (i = 0; i < number of threads; i++) {
        pthread_join(thread ID, NULL);
    }

}
```

Genome= CACCATAAAACCGTGTGGGA

Sequence= CCGT

Length of Genome = 20

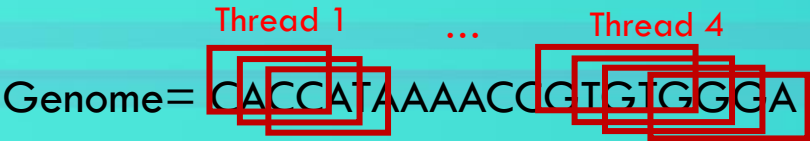
Length of Sequence = 4

Number of threads = 4

Steps = ?

Steps = number of tests / number of threads

Number of tests = Length of Genome – Length of Sequence



Steps = 16 / 4 = 4

Number of tests = 20 – 4 = 16

```
int main() {

    printf("Insert the genome filename:");
    //Recieve the genome file name from the user

    printf("Insert the sequence filename:");
    //Recieve the sequence file name from the user

    // read the files and save them in a string
    fgenome = fopen(genome_filename, "r");
    // read the files and save them in a string
    fseq = fopen(seq_filename, "r");

    printf("Insert the number of threads:");
    //Recieve the number of threads from the user

    /* Allocate an array to store the parms of all created threads*/

    // Define ID of threads
    pthread_t ....
    // Define struct of each thread
    struct thread_parms ...

    for (i = 0; i < number of threads; i++) {

        /* Prepare the thread parameters */
        //The start point where thread should start to search for the sequence = start

        // The End point where thread should stop to search for sequence= Stop
        Stop = start + Step;

        //Create the thread
        err = pthread_create(&thread ID, NULL, &thread Fucntion, (void*)& thread_struct);
        start += step;
    }
    /* wait for all thread to finish */
    for (i = 0; i < number of threads; i++) {
        pthread_join(thread ID, NULL);
    }

}
```


❑ Exercises 1:

Steps:

1. Create the two files (Genome and sequence)
2. Receive the name of the files by the program, open and read them and save them in a string.
3. Receive the number of the threads from the user.
4. Generate the requested number of threads.
5. Calculate the start and stop search position for each thread.
6. Save the start and stop in the structure and send it to thread.
7. Write a function which is managing the search of the sequence is a segment dedicated to that thread.
8. Wait in the master threads for other threads to finish.

```
int main() {

    printf("Insert the genome filename:");
    //Recieve the genome file name from the user

    printf("Insert the sequence filename:");
    //Recieve the sequence file name from the user

    // read the files and save them in a string
    fgenome = fopen(genome_filename, "r");
    // read the files and save them in a string
    fseq = fopen(seq_filename, "r");

    printf("Insert the number of threads:");
    //Recieve the number of threads from the user

    /* Allocate an array to store the parms of all created threads*/

    // Define ID of threads
    pthread_t ....
    // Define struct of each thread
    struct thread_parms ...

    for (i = 0; i < number of threads; i++) {

        /* Prepare the thread parameters */
        //The start point where thread should start to search for the sequence = start

        // The End point where thread should stop to search for sequence= Stop
        Stop = start + Step;

        //Create the thread
        err = pthread_create(&thread ID, NULL, &thread Fucntion, (void*)& thread_struct);
        start += step;
    }
    /* wait for all thread to finish */
    for (i = 0; i < number of threads; i++) {
        pthread_join(thread ID, NULL);
    }

}
```

❑ Exercises 2:

Write a multi-thread program to search if a number N is a prime number by checking if it is divisible by any number in the range $[2, N/2]$. The program receives in input the number N and the number P of threads to use. The program must create P threads and assign to each of them a subset of possible divisors to test. Each thread checks if N is divisible by any of the assigned divisors. The main program continuously checks the result of the generated threads. As soon as a thread detects that the number is not prime, the main program must stop the executions of all generated threads and exit.

Once the program is ready, try its execution with big numbers and different values of P to understand how the execution time changes. Printing the start and the end time is a good idea to trace the whole execution time.

```
struct thread_parms {  
    int N;  
    int start;  
    int stop;  
};
```

```
void* runner(void* param) {  
  
    struct thread_parms* p = (struct thread_parms*) param;  
    /* tests all divisors of the assigned interval */  
    for (i = p->start; i <= p->stop ; i++) {  
        ....  
        ....  
        ....  
    }  
}
```

```
int main() {  
  
    printf("Insert the number to test:");  
    scanf("%d", &N);  
  
    printf("Insert the number of threads:");  
    scanf("%d", &P);  
  
    pthread_t ...;  
    struct thread_parms ...;  
  
    step = number of test / number of threads;  
  
    for (i = 0; i < number of threads; i++) {  
  
        /* Prepare the thread parameters */  
        thread_struct.[Start]  
        thread_struct.[Stop]= thread_struct.[Start] + Step  
  
        //Create the thread  
        err = pthread_create(&tid, NULL, &thread Function, (void*)& thread_struct);  
  
        ....  
    }  
}
```