Multithreaded Programming

Dr. H. İrem Türkmen

Outline

- Process and Threads
- Multithread Programming
- Multithread Programming in C
- Pros and cons of multithreaded programming
- Pthread

Process and Threads

- **A process** is an independently running instance of a program.
- Each process maintains its own heap, stack, registers and file descriptors
- **A thread** is smallest sequence of program instructions that shares its memory space with others.
- A process can have multiple threads of execution.

Process vs. Threads

- Processes do not share their memory space, while threads executing under same process share the memory space.
- Processes execute independent of each other and the synchronization between processes is taken care by kernel only; on the other hand, thread synchronization has to be taken care by the process under which the threads are

Process vs. Threads

- Processes have independent open file descriptors, while threads have shared open file descriptors
- The interaction between 2 processes is achieved only through the standard inter-process communication, while threads executing under the same process can communicate easily as they share most of the resources like memory, text segment etc

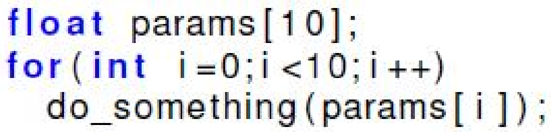
Multithreaded programming

- Serial execution:
  - All our programs so far has had a single thread of execution: main thread.
  - Program exits when the main thread exits.
- Multithreaded:
  - Program is organized as multiple and concurrent threads of execution.
  - The main thread *spawns multiple threads*.
  - The thread **may communicate with one another.**

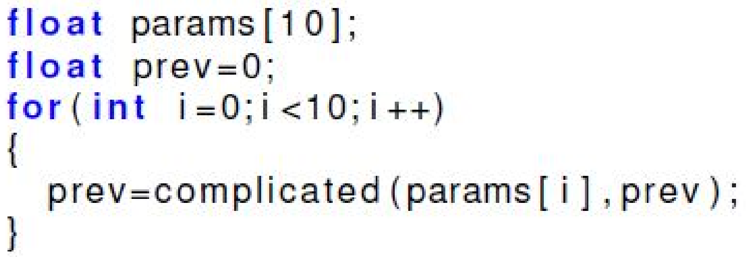
Multithreaded programming in C

- Pthreads: POSIX C library.
- OpenMP
- Intel threading building blocks
- Grand central despatch
- CUDA (GPU)
- OpenCL (GPU/CPU)

Not all code can be made parallel



```
float params[10];
for(int i=0;i<10;i++)
  do_something(params[i]);
```



```
float params[10];
float prev=0;
for(int i=0;i<10;i++)
{
  prev=complicated(params[i],prev);
}
```

Pros and cons of multithreaded programming

- Advantages:
  - Improves responsiveness
  - Improves utilization
  - Less overhead compared to multiple processes
- Disadvantages:
  - Debugging with threads is difficult.
  - Too many threads may reduce the performance.

Creating a Thread

- #include <pthread.h>
- Define a thread ID
  - A Thread ID is unique in the context of current process.
  - It could be a structure and represented by type pthread_t
  - pthread_t tid;

Creating a Thread

- Define set of thread attributes
  - pthread_attr_t : type that contains the attributes of a thread object (stack address, stack size, scheduling parameters etc.)
  - pthread_attr_t attr;

    pthread_attr_init(&attr); function initializes the thread attributes object pointed to by *attr* with default attribute values.
  - After this call, individual attributes of the object can be set using various related functions
  - NULL can be used to create a thread with default arguments

Creating a Thread

- Define a worker function
- This function contains the code segment which is executed by the thread.
  - void * foo (void *args)

    {

        …

        …

    }

Creating a Thread

- pthread_create() function in pthread.h file, is used to create a thread.
- The syntax and parameters details are given as follows:
  - int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void * (*start_routine) (void *), void *arg);
  - If thread created successfully, return value will be 0 otherwise pthread_create will return an error number of type int.

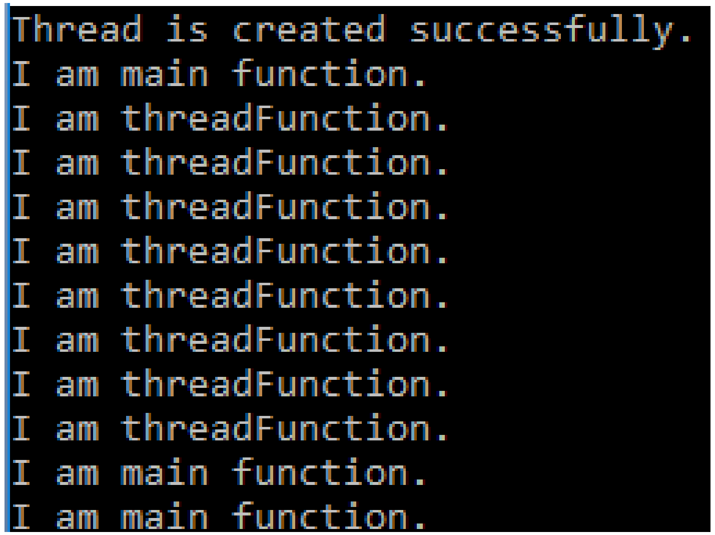
How to compile & execute?

- gcc filename.c -o outputfilename –lpthread
- ./outputfilename

```
#include <stdio.h>
#include <pthread.h>
/*thread (worker) function definition*/
void* threadFunction(void* args){
    while(1)
        printf("I am threadFunction.\n");
}

int main()
{
 /*creating a thread id in the main function*/
 pthread_t id;
 int ret;
 /*creating thread*/
 ret=pthread_create(&id, NULL, threadFunction, NULL);
 if(ret==0)
    printf("Thread is created successfully.\n");
 else
 {
    printf("Thread is not created.\n");
    return 0; /*return from main*/
 }

 while(1)
    printf("I am main function.\n");
 return 0;
}
```

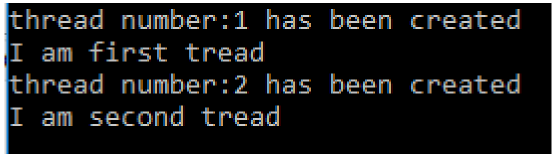


```
Thread is created successfully.
I am main function.
I am threadFunction.
I am threadFunction.
I am threadFunction.
I am threadFunction.
I am threadFunction.
I am threadFunction.
I am threadFunction.
I am threadFunction.
I am main function.
I am main function.
```

```c
#include<stdio.h>
#include<pthread.h>
#include<unistd.h>
pthread_t tid[2];
void* worker(void *arg)
{
    pthread_t id = pthread_self();
    if(pthread_equal(id,tid[0]))
        printf("I am first tread\n");
    else
        printf("I am second tread\n");

}
int main(void)
{
    int i = 0;
    int err;
    for (i=0;i<2;i++)
    {
        err = pthread_create(&(tid[i]), NULL, worker, NULL);
        if (err != 0)
            printf("can't created\n");
        else
            printf("thread number:%d has been created\n",i+1);

    }
    sleep(5);
    return 0;
}
```



```
thread number:1 has been created
I am first tread
thread number:2 has been created
I am second tread
```

pthread_join()

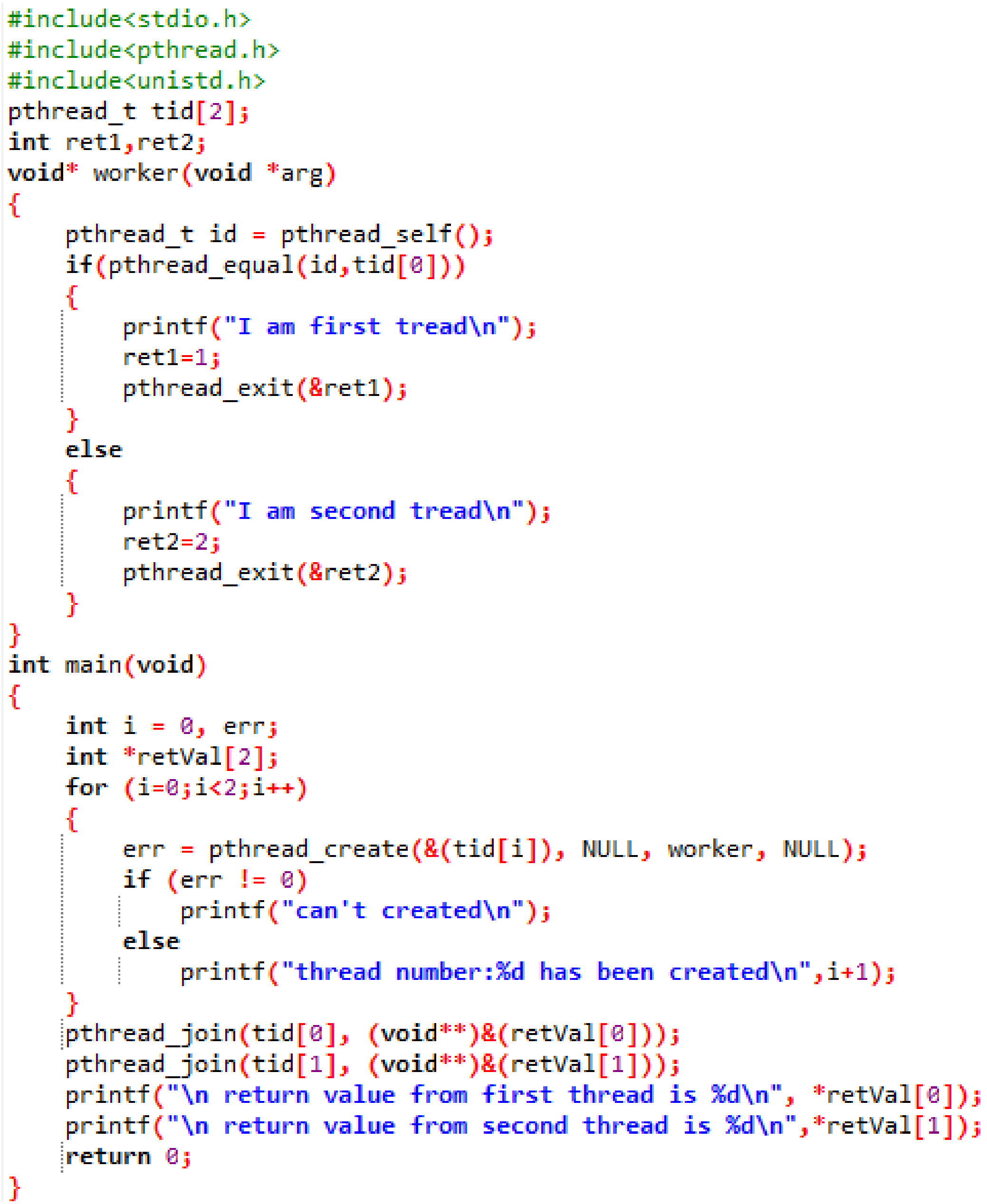
- Without the sleep() function, we did not see the message of “I am second tread”.
- Just before the second thread is about to be scheduled, the parent thread, from which the two threads were created, completed its execution.
- To make main function to wait for each thread to complete: pthread_join()

pthread_join()

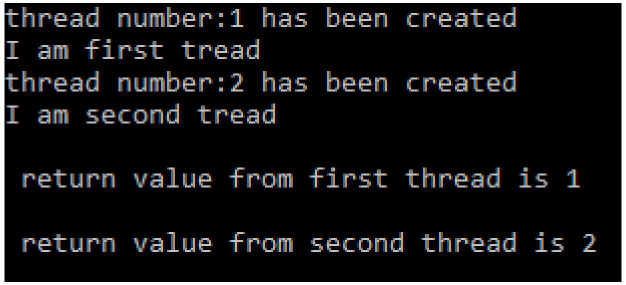
- int pthread_join(pthread_t *thread*, void ***retval*);
- The pthread_join() function waits for the thread specified by *thread* to terminate.
- If we are not interested in the return value then we can set this pointer to be NULL.
- If *retval* is not NULL, then pthread_join() copies the exit status of the target thread.

pthread_exit()

- void pthread_exit(void **retval*);
- Terminates the calling thread and returns a value via *retval* that (if the thread is joinable) is available to another thread in the same process that calls pthread_join()



```c
#include<stdio.h>
#include<pthread.h>
#include<unistd.h>
pthread_t tid[2];
int ret1,ret2;
void* worker(void *arg)
{
    pthread_t id = pthread_self();
    if(pthread_equal(id,tid[0]))
    {
        printf("I am first tread\n");
        ret1=1;
        pthread_exit(&ret1);
    }
    else
    {
        printf("I am second tread\n");
        ret2=2;
        pthread_exit(&ret2);
    }
}
int main(void)
{
    int i = 0, err;
    int *retVal[2];
    for (i=0;i<2;i++)
    {
        err = pthread_create(&(tid[i]), NULL, worker, NULL);
        if (err != 0)
            printf("can't created\n");
        else
            printf("thread number:%d has been created\n",i+1);
    }
    pthread_join(tid[0], (void**)&(retVal[0]));
    pthread_join(tid[1], (void**)&(retVal[1]));
    printf("\n return value from first thread is %d\n", *retVal[0]);
    printf("\n return value from second thread is %d\n",*retVal[1]);
    return 0;
}
```



```
thread number:1 has been created
I am first tread
thread number:2 has been created
I am second tread

 return value from first thread is 1

 return value from second thread is 2
```

Void pointer in C

- A void pointer is a pointer that has no associated data type with it.
- It can hold address of any type and can be casted to any type.
- Advantages of the void pointers:
    - Void pointers in C are used to implement generic functions (e.g. qsort() function)
    - Functions such as malloc(), calloc() return void* type. Hence, they can allocate memory for any data type (just because of the void *)

Void pointer in C

- Standard C does not allow pointer arithmetic with the void*. However, GNU C considers the size of the void is 1 byte.
- void * cannot be derefenced. The use in the left-side is illegal:

```
int a = 10;
void *ptr=&a;
printf("%d", *ptr); // ILLEGAL
printf("%d", * (int*) ptr); // LEGAL
```